MORITZ LIPP

Exploiting Microarchitectural Optimizations from Software

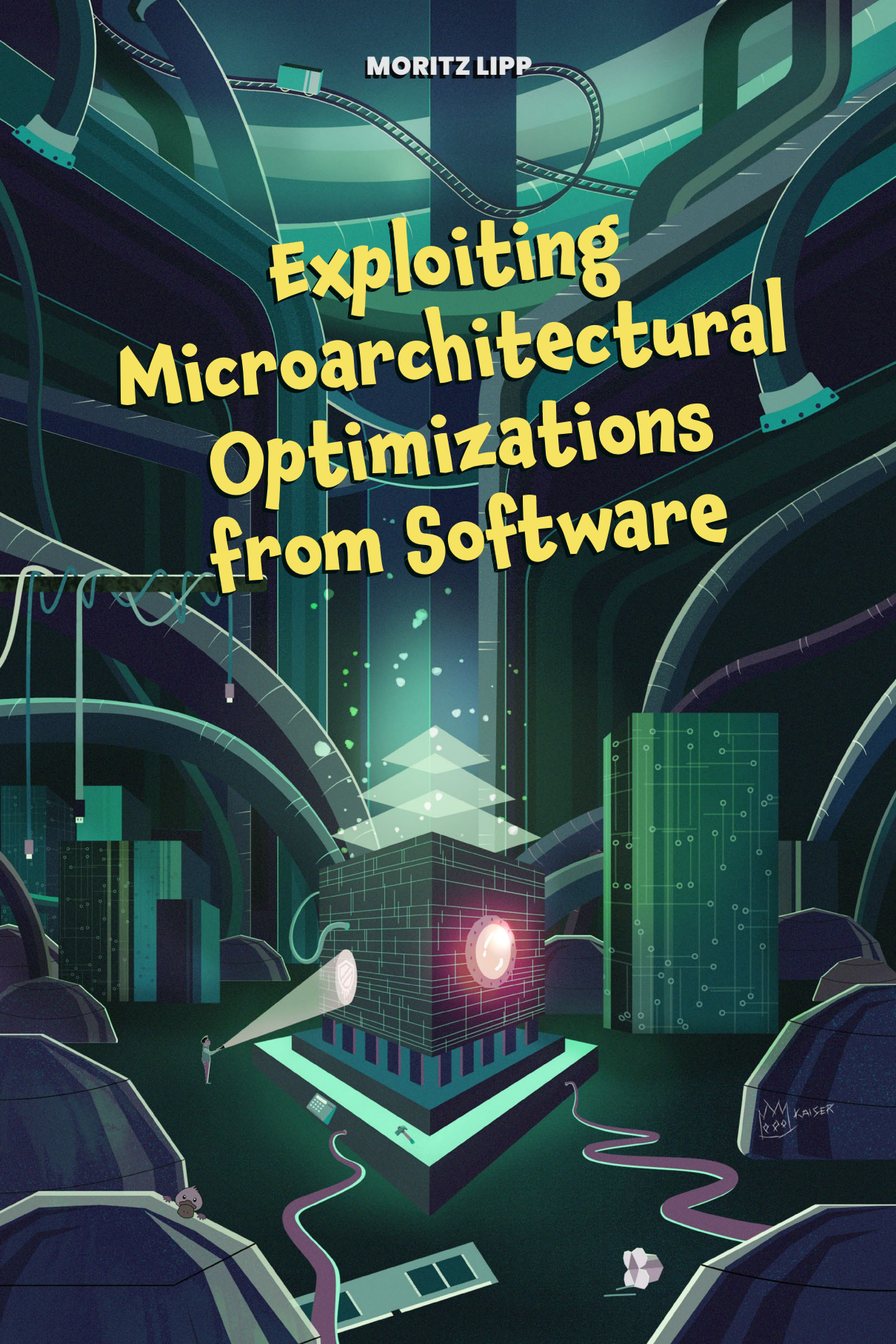


Illustration
Natascha Eibl

Exploiting Microarchitectural Optimizations from Software

by

Moritz Lipp

Ph.D. Thesis

Assessors

Daniel Gruss (Graz University of Technology)
Thomas Eisenbarth (University of Lübeck)

August 2021

Institute for Applied Information Processing and Communications
Faculty of Computer Science

Graz University of Technology

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date, Signature

Abstract

With abstraction layers, the implementation details of software and hardware components are hidden away to deal with the complexity of modern computer systems. While the Instruction Set Architecture (ISA) serves as an interface between the CPU and the software running on it, the computer microarchitecture is the actual hardware implementation of the ISA. The clearly defined interfaces do not only cover up the complexity but also allow different variants of the microarchitecture to be built. While they all fulfill the contract defined by the ISA, they can differ in other aspects, such as performance, security, energy efficiency, or other physical properties. Microarchitectural attacks exploit these variations occurring on the microarchitectural level of modern CPUs. With side-channel attacks and fault attacks, there are different ways that allow learning from and tampering with the actual implementation. These attacks allow adversaries to extract sensitive information processed on the system, e.g., cryptographic keys or user behavior.

In this thesis, we expand the landscape of software-based microarchitectural attacks and defenses. By exploring the security implications of different optimizations, we identify previously unknown attack vectors, allowing us to circumvent the most fundamental security guarantees of modern processors. We combine traditional physical side-channel analyses with software-based microarchitectural attack techniques to leak sensitive information processed on the CPU. We enlarge our understanding of which settings and circumstances facilitate different existing attacks and give new insights into developing effective and efficient mitigations.

In the first part of this thesis, we discuss the contributions of this thesis and provide background on CPU architecture and memory organization, as well as side-channel attacks and fault attacks. Furthermore, we discuss the state of the art of software-based microarchitectural attacks and defenses. In the second part, a selection of my peer-reviewed publications is provided without modification from their original versions.

Acknowledgements

First and foremost, I want to thank my advisor, long-time office colleague and friend, Daniel Gruss, who initially sparked my keen interest in microarchitectural attacks. While I have always considered side-channel attacks as black magic and, therefore, have been very reluctant to learn about them, working with Daniel on my master thesis ignited my fascination for this research field. I want to thank you for your continuous support, the freedom you granted me in my research, the endless effort you invest in everything, and your openness for different opinions.

I would especially like to thank Michael Schwarz for his friendship, tireless support, and ambition to work on all those different projects with me. I really enjoyed our time working together, discussing all kinds of ideas while keeping the caffeine level at a healthy high, and discovering parts of the world after conferences. I will miss the late-night debugging sessions, ambitious paper sprints, and presenting our work together.

Looking back over the last years, I want to thank you both, Daniel and Michael, for this incredible journey. I really enjoyed sharing the office with you, all the discussions, and the endless effort we put in as a team. The ups and downs we faced together make this roller coaster ride unforgettable for me.

I want to thank Thomas Eisenbarth for valuable feedback, interesting discussions and taking the time and effort to assess my thesis.

Furthermore, I would really like to thank Stefan Mangard for giving me the opportunity to pursue a Ph.D. in the first place. Thank you for your support and advice over the years.

Over the last couple of years, I had the honor to meet, work and make friends with incredibly kind and talented people worldwide. While the list would be endless, I want to especially thank Jo van Bulck and Daniel Moghimi for insightful discussions and the great teamwork. Likewise, I want to thank Anders Fogh, David Oswald, Berk Sunar, Julian Stecklina, and Thomas Prescher for great discussions and fruitful collaborations. I am further grateful to all my (former) colleagues at the institute for insightful discussions and for creating such an enjoyable working environment, in particular, Martin Schwarzl, Claudio Canella, Lukas Giner, Catherine Easdon, Andreas Kogler, Clémentine Maurice, Robert

Schilling, Mario Werner, Peter Pessl, Stefan More and Sebastian Ramacher.

I want to thank my parents, Peter and Michaela, and my siblings, Ilona, Lukas, and Nikolaus, for all their love and support throughout my entire life. I would like to thank my loving family, Gertraud, Eva, Helga, Gerhard, Heidrun, and Renate, for their support.

A special thanks goes to all my friends; without their support, I could not have completed this thesis. Thank you for meaningful discussions, even late at night, and providing the necessary distractions to the working life. Furthermore, I want to thank Yuki, Rasputin, and my bees for always helping me to find tranquilness even in the most stressful times.

Finally, I want to be grateful to my better half, Natascha, for her never-ending supporting love and patience. Thank you for tolerating all *following deadlines* and supporting me throughout all those years. This work could not have been done without you.

Contents

Part I.

Exploiting Microarchitectural Optimizations from Software

1

Introduction and Contribution

In software engineering and computer science, interfaces are defined boundaries that are shared between two or more components enabling them to interact with each other. These components exist in software and hardware, can be a combination of both, or in fact, even peripheral devices or human beings interacting with a device [108]. By specifying how components are supposed to interact with each other, what the expected behavior of the other component is, interfaces grant great flexibility in the implementation of the component as long as the expected behavior is maintained.

By abstracting away components of a system, simple interfaces do not only cover up the complexity of the system but allow different variants of a component to be implemented. For example, in a software project, a new, more performant algorithm is implemented for a defined interface. Conforming to the interface's specification, this allows it to easily replace the old implementation without the need to modify the rest of the system. However, these interfaces do not only exist within software projects but also between applications and the operating system or between developers and the actual hardware. While the implementations all fulfill the same purpose, they can differ in other aspects like the performance.

Architecture vs. Microarchitecture

In computing, an architecture typically refers to the instruction set architecture (ISA), also called computer architecture. While the ISA serves as an interface between the CPU and the software running on it, the computer microarchitecture is the actual hardware implementation of the architecture [103]. Thus, the complexity of the implementation is abstracted away by the (less complex) architecture specification. With different microarchitectures implementing the same architectural specification, the same application can perform differently on them, *i.e.*, while

the computed results are the same, the runtime performance, energy efficiency, or other physical properties may vary. However, this view is not limited to the central processing unit alone but can be applied to every abstraction layer. This includes other hardware interfaces, e.g., the DRAM, as well [154, 267]. As long as the memory module conforms to the specification, the manufacturer can carry out its implementation in different ways. Furthermore, this view is not limited to hardware alone. For instance, the operating system provides an interface for user space applications to interact with it, *i.e.*, system calls and signals. Thus, different versions of an operating system can be seen as different microarchitectures as well, as the internal workings of the operating system might differ while the functional correctness remains.

Optimizing Behind Closed Doors

In the past, modern processors have been solely optimized for performance and power consumption. As end customers demand faster and faster processors, manufacturers have to improve and optimize their processors as much as possible. In a benchmark-defined world, every cycle counts and plays a role in the market shares of manufacturers. Moreover, operating systems and user-space applications try to get the most out of the platform they are running on.

One typical way to increase the performance is to optimize for the common case, which is the case that occurs the most frequently. If the common case how an interface is used can be handled more efficiently, forming a fast path of its execution, than uncommon cases (like corner cases or runtime errors), it results in a better performance.

The clear abstraction boundaries defined by interfaces enable us to *do whatever we want* in the underlying implementation as long as the behavior visible to the user of the interface meets the definition of the interface. This grants developers endless possibilities of implementing the interface and, thus, how they can optimize the microarchitecture for different factors.

There are many different ways and locations that allow improving the performance of an application running on a system: With *compiler optimizations*, the source code is transformed and optimized for different attributes [200]. With *runtime optimizations*, different interpreters, e.g., JavaScript engines [291], or other runtime environments, e.g., Java vir-

tual machines [301], optimize the program during runtime depending on the workload. The *operating system* the application is running on can be tuned to not only give an application more resources but also in the way these resources and the communication from the application is handled. For instance, one of the main tasks of the operating system is to manage the memory of the system. To do that, it needs to create the necessary structures for processes to assign segments of the memory. While several optimization techniques, e.g., copy-on-write, demand paging, swapping or memory deduplication, allow optimizing the memory consumption of applications and increase the performance of the system, they are not visible to the user space application itself [281]. The *hardware*, *i.e.*, the processor, that executes the software allows for optimizations. Besides the CPUs clock rate, there are many factors that influence the performance of the processor. Different sizes and properties of caches play a role, as well as if the processor has out-of-order or in-order execution [268]. Furthermore, the use of various predictors can increase the performance drastically [103].

Thus, in order to speed up the overall performance of the system, security guarantees of the architecture have often been ignored on a microarchitectural level. As the inner workings of the microarchitecture are hidden and cannot be inspected from the outside, and as long as these assumptions are guaranteed to hold on an architectural level, the systems behave as expected.

In this thesis, we focus on the security implications these types of optimizations can yield.

Shining light through the opaque

Clearly, defining interfaces and abstracting away complexity led us to believe that the implementation appears to be an opaque black box and that its inner workings cannot be inspected from the outside. This assumption even allows to ignore security guarantees on the microarchitectural level in order to improve the performance [105, 244, 272]. However, information can not only be transmitted over *legitimate channels* by the specified interfaces, but an implementation can reveal additional information over so-called *incidental channels* [127], e.g., the response time or the energy consumed by the implementation. These channels serve as *side channels* if the victim leaks information over this channel, enabling *side-channel attacks* if the leaked information can be exploited by an adversary.

With *side-channel attacks* and *fault attacks*, there are non-invasive and respectively invasive ways allowing to learn from and tamper with the actual implementation of a device to deduce sensitive information [184]. While these attacks typically require physical access to the hardware, we want to focus on similar techniques, however, by mounting attacks against these interfaces from software only.

Side-Channel Attacks. With the assumption that the inner workings of the microarchitecture can not be inspected, side-channel attacks are usually not taken into account in the processor's threat model. Side-channel attacks exploit information leakage of a system's implementation in hardware and software. Typically, physical properties like power consumption or magnetic radiation are monitored, and the obtained measurements are used to deduce otherwise inaccessible information. As an example, when executing a cryptographic operation, in some algorithms, the processor has to perform a more power-consuming operation when processing a '1' key-bit than when processing a '0' [254]. By monitoring the power consumption, an attacker can correlate the power trace with the operations performed and, thus, recover the entire key [184]. Here, the attacker observes indirect information about the key, *i.e.*, metadata. More specifically, with knowledge of the power consumption of specific operations, this information can be mapped to the key, but the key cannot be read out directly.

Historically, side-channel attacks required physical access to the target under attack to either connect probes or other peripherals to perform the measurements [184]. However, in recent years, many different software-based attacks surfaced, thus, lifting the requirement of physical access [275]. Many microarchitectural side-channel attacks target the cache of the processor. As a transparent optimization technique in the microarchitecture, caches are small but fast memory, allowing the processor to access recently-used data faster. The timing difference introduced by data being cached or not allows an attacker to build software-based side-channel attacks. These so-called cache attacks allow attacking cryptographic algorithms [287, 334, 335], monitor user behavior [94, 174, 220], spy on virtual machines [114, 131], or attack ASLR [79] and kernel ASLR (KASLR) [89, 110, 140, 320].

Fault Attacks. With fault attacks, an adversary intentionally brings devices for a short time into physical conditions which are outside the device's specification. This can be achieved by temporarily using incorrect supply voltages, exposing the device to high or low temperature, radiation, or by dismantling the chip and shooting at it with lasers. If software can bring the device to the border or outside of the specified operational conditions, software-induced hardware faults are possible [153, 282]. With the Rowhammer bug, Kim et al. [153] demonstrated a hardware reliability issue in DRAM where repeatedly accessing a specific memory location flips bits in physically adjacent memory locations. These bit flips have been exploited to obtain arbitrary write primitives [266], root privileges [91, 300] or to read inaccessible memory locations [161]. CLKSCREW [282] exploited software-exposed energy management mechanisms to induce faults during computations on ARM-based devices. By undervolting the CPU [53, 150, 202, 203, 230–232] through a software interface, faulty computations allow to leak sensitive data.

1.1. Main Contributions

The main contributions of this thesis demonstrate the security implications of various microarchitectural optimizations by exploiting them through software-accessible interfaces.

The contributions of this thesis *advance the state of the art of microarchitectural attacks and defenses* by:

- **Discovering *Transient-Execution Attacks.*** With Meltdown (Chapter 6), we exploit the transient execution of instructions before a fault is actually handled in out-of-order CPUs, while with Spectre [156], we exploit the transient execution of instructions caused by mispredictions. This attack does not only bypass the most fundamental security guarantees of modern processors by circumventing memory isolation, but it allows, in contrast to classical side-channel attacks, to leak data processed on the machine *directly*. With Meltdown and Spectre, a whole new research field emerged in the area of microarchitectural attacks, namely transient-execution attacks.
- **Identifying *previously unknown attack vectors.*** With Takeaway (Chapter 5), we reverse-engineer the cache way predictor of AMD CPUs and present two new attack techniques leaking metadata to recover sensitive information.

- **Exploring if different existing attacks can be mounted *remotely*.** With Nethammer (Chapter 7), we show that Rowhammer faults can be induced on commodity hardware through network requests alone, enabling adversaries to induce bit-flips without any control over the code executed on the target machine. We show that the interrupt-based attack vector described in KeyDrown can also be mounted from JavaScript within a browser, enabling a remote way to observe inter-keystroke timings. By targeting the interrupt handling, we observe inter-keystroke timings of a user's PIN or password in sandboxed JavaScript and infer URLs entered by a user on personal computers and smartphones (Chapter 8).
- **Combining *traditional physical side-channel analysis* with *modern software-based microarchitectural attack techniques*.** With PLATYPUS (Chapter 9), we exploit the processor's energy consumption available to software to infer data and extract cryptographic keys. While the update interval of the interface is low compared to traditional oscilloscopes used for physical side-channel attacks, we use techniques from microarchitectural attacks to control the execution of SGX enclaves and overcome these limitations to extract cryptographic keys.
- **Giving *new insights* into efficiently mitigating attacks.** Discovering new attack vectors and studying existing ones in more depth allows us to better understand the requirements for efficient mitigations. With KAISER (Chapter 10), we initially proposed a stronger page-table isolation for operating systems to mitigate various side-channel attacks on KASLR. In addition, it turned out that the design of KAISER offered a software-only defense to protect against the Meltdown attack and, thus, has been adopted in every major operating system.

The rest of this section briefly describes the individual contributions of this thesis, while Part II includes the corresponding peer-reviewed publications. However, as only a subset of the contributions conducted during my Ph.D. is included within this thesis, a brief description of other results can be found in Section 1.2.

Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. To optimize the energy consumption and performance of their processors, AMD deploys a way predictor for the L1 data cache

to predict in which way of the cache a certain address is stored. The implementation tags each cache line with a linear-address-based micro-tag that is computed using an undocumented hash function. In this work, we first reverse-engineer this hash function in microarchitectures from 2011 to 2019 and present two new attack techniques using the way predictor as a side channel: Collide+Probe and Load+Reload. Based on these techniques, we demonstrate a covert channel, reduce the entropy of ASLR on the kernel (KASLR) and in the browser, recover cryptographic keys and exfiltrate secret data from the kernel using a Spectre attack.

The paper "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors" was published at *AsiaCCS* 2020 in collaboration with Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss.

Meltdown: Reading Kernel Memory from User Space. Under the assumption that the microarchitectural state is invisible and that microarchitectural state changes cannot be observed, security guarantees could be ignored during transient execution as they have no visible consequences. As microarchitectural side-channel attacks have been limited to leak only metadata about the execution of a program, *i.e.*, executed instructions or data accesses, they are usually not taken into account in the processor's threat model. With Meltdown, however, we exploit that during transient execution, the permission check required by the page tables is deferred on some microarchitectures. Thus, architecturally inaccessible data is forwarded to the adversary in the transient domain. Using a microarchitectural covert channel, the adversary can transmit the data to the architectural domain leaking arbitrary memory.

Together with Spectre [156], Meltdown describes a new class of microarchitectural attacks and paved the way for a variety of transient-execution attacks and their necessary mitigations. With Foreshadow [293], Zombieload [262], RIDL [251], Fallout [46], CrossTalk [236], LazyFP [276], CacheOut [248, 252], and Medusa [197] many other Meltdown-type attacks followed, highlighting the importance and research interest in the new field of transient-execution attacks. We discuss the state of the art and, thus, the immense research efforts that followed our discovery in Section 3.2.

The paper "Meltdown: Reading Kernel Memory from User Space" was published at *USENIX Security Symposium* 2018 in collaboration with

Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg.

Practical Keystroke Timing Attacks in Sandboxed JavaScript. To mount a keystroke timing attack, an attacker so far either requires physical access to the device under attack [199], or local code execution [94, 199, 261]. We investigated whether the interrupt-based attack vector described in our paper "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks" [261] can also be mounted from JavaScript within a browser. We showed that we could not only successfully spy on keystrokes on desktop machines but also on touches and swipes on mobile phones. We showed that an attacker could distinguish between website URLs a user entered in the browser bar and between users time-sharing a machine.

The paper "Practical Keystroke Timing Attacks in Sandboxed JavaScript" was published at *ESORICS* 2017 in collaboration with Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard.

KASLR is Dead: Long Live KASLR. As recommended by Intel [118], the kernel should be mapped into the address space for every user application. However, for kernel pages, the userspace-accessible bit in the page tables is not set. Therefore, the address space is separated into virtually two; one for user mode and one for kernel mode.

However, this single mapping enabled various side-channel attacks breaking KASLR [89, 110, 140]. In 2016, we already proposed a change to operating systems to mitigate prefetch side-channel attacks [89] that sets up two mappings, one for the kernel and one for the user space application. We implemented this idea as a proof-of-concept patch for the Linux kernel and evaluated that it successfully impedes the attacks. In the meantime, the stricter isolation provided by KAISER proved itself useful as it also protects against the Meltdown attack [177] and, thus, has been adopted in every major operating system by now.

The paper "KASLR is Dead: Long Live KASLR" was published at *ESSoS* 2017 in collaboration with Daniel Gruss, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard.

Nethammer: Inducing Rowhammer Faults through Network Requests. The Rowhammer attack has always been considered a local attack where the attacker requires access to the machine to execute code or has to trick a victim into accessing a malicious website that induces bit flips via JavaScript [88]. With *one-location hammering* [88], the question arises whether it is possible to trigger bit flips remotely over the network without the execution of attacker-controlled code. By simply crafting minimal packets and sending them as fast as possible to the victim machine, we showed that bit flips can be induced remotely on commodity hardware. We demonstrated a series of remote attacks leading to temporary or persistent errors in the system. Furthermore, we demonstrated that the hardware countermeasure target-row-refresh (TRR) is insufficient to protect against local and remote Rowhammer attacks.

The paper "Nethammer: Inducing Rowhammer Faults through Network Requests" was published at the *SILM Workshop* 2020 in collaboration with Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss.

PLATYPUS: Software-based Power Side-Channel Attacks on x86. In a classical power side-channel attack setting, an adversary requires physical access to the device to monitor the energy consumption using an oscilloscope. To remain within power constraints, CPU vendors provide an interface to internal power meters allowing to obtain the energy consumption of the core, main memory, or the entire package. On Intel CPUs, the Intel Running Average Power Limit (RAPL) grants software access to these energy measurements. While the update interval of RAPL is rather low in contrast to real oscilloscopes, we show that with sufficient statistical evaluation, we can observe variations in the power consumption. This does not only allow us to distinguish between different instructions but also between different Hamming weights of operands and memory loads.

With PLATYPUS, we exploit the unprivileged access to this interface to leak AES-NI keys from Intel SGX and the Linux kernel, break KASLR and establish a timing-independent covert channel. Furthermore, we leverage tools from microarchitectural attacks to precisely control the execution of an Intel SGX enclave in combination with the RAPL interface to recover RSA keys processed within the enclave.

The paper "PLATYPUS: Software-based Power Side-Channel Attacks on x86" was published at *IEEE S&P* 2021 in collaboration with An-

dreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss.

1.2. Other Contributions

In this section, we want to briefly discuss other publications I have contributed to during my Ph.D. that are not included within this thesis.

KeyDrown. While in many cases where the implementation of a cryptographic algorithm can be attacked with a side-channel attack, the actual implementation can be protected by modifying the code or switching to other implementation primitives. However, protecting against attacks monitoring user input by observing keystrokes is more difficult. Our idea to mitigate such attacks is to artificially generate fake interrupts that introduce indistinguishable behavior in the observed side channel. User keystrokes are treated as noise on their own and are blended in the noise of the generated artificial keystrokes such that an attacker cannot distinguish between real and fake key strokes. While introducing noise as a countermeasure against side-channel attacks usually increases the number of measurements required to mount the attack successfully, the introduced noise renders keystroke attacks in this scenario infeasible.

The paper "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks" was published at *NDSS* 2018 in collaboration with Michael Schwarz, Daniel Gruss, Samuel Weiser, Clementine Maurice, Raphael Spreitzer, and Stefan Mangard.

JavaScript Zero. With "Practical Keystroke Timing Attacks in Sandboxed JavaScript" [173], we proposed a fine-grained permission system in JavaScript for browsers to mitigate browser-based side-channel attacks. Typically, attacks in web browsers exploit primitives that are either rarely used or used in an unintended way. We checked how many of the primitives that have been used to mount side-channel attacks or exploits in browsers are used on the 10 most popular websites [14]. We figured out that many of them are not used at all. While some sites rely on some features, they do not necessarily rely on the high resolution a feature provides. The idea of this work is to introduce a dynamic approach that removes these features or modifies their behavior in a way that mitigates the attacks but does not influence the usability of the websites.

The paper "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks" was published at *NDSS* 2018 in collaboration with Michael Schwarz and Daniel Gruss.

Another Flip. The Rowhammer bug [153] is a disturbance error in DRAM where repeatedly accessing a memory location leads to bit flips in physically adjacent locations. This hardware issue can be triggered from software and, thus, has been exploited in various attack settings [69, 91, 179, 299]. In this work, we propose a new hammering technique where we repeatedly only access a single memory location. Furthermore, we showed that using this technique in combination with Intel SGX and by inducing bit flips in the program code; we can circumvent all existing Rowhammer defenses.

The paper "Another Flip in the Wall of Rowhammer Defenses" was published at *IEEE S&P* 2018 in collaboration with Daniel Gruss, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom.

Use-After-FreeMail. Another class of attacks is use-after-free attacks, where an application attempts to access a recently freed memory location. We generalized these attacks and demonstrate that they can be applied to different settings, e.g., to email addresses.

The paper "Use-after-freemail: Generalizing the use-after-free problem and applying it to email services" was published at *AsiaCCS* 2018 in collaboration with Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, and Stefan More.

Double-Fetch Bugs. Double-fetch bugs are an exploitable race condition where a privileged context accesses an unprivileged external resource multiple times. If the content of the external resource can be modified in between accesses, the other access to a now different value can be exploitable, e.g., time-of-check to time-of-use (TOCTTOU). We showed that cache side channels allow detecting such bugs in code and that they can be used as a trigger signal, outperforming state-of-the-art exploitation techniques. Furthermore, we showed that Intel TSX can be used to mitigate the exploitation of double-fetch bugs automatically.

The paper "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features" was published at *AsiaCCS* 2018 in collaboration with Michael Schwarz, Daniel Gruss, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard.

Spectre. Simultaneously to our work on Meltdown [177], we identified issues with speculative execution in modern processors caused by the mispredictions of branch predictors in the CPU. We showed that we can influence the predictors to speculatively execute code that would architecturally never be executed and, consequently, leak sensitive data accessed by the victim.

The paper "Spectre Attacks: Exploiting Speculative Execution" was published at *IEEE S&P* 2019 in collaboration with Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom.

NetSpectre. With the same question as with Nethammer [179], we wanted to investigate if it is possible to exploit Spectre attacks over the network. We demonstrated how an attacker can leak data remotely over a network-accessible API by measuring the response time of network packets. Furthermore, we presented a novel side channel abusing Intel AVX instructions.

The paper "NetSpectre: Read Arbitrary Memory over Network" was published at *ESORICS* 2019 in collaboration with Michael Schwarz, Martin Schwarzl, Jon Masters, and Daniel Gruss.

A Systematic Evaluation of Transient Execution Attacks and Defenses. After Meltdown [177] and Spectre [156], a new research field of transient-execution attacks opened up. After the disclosure of Foreshadow [293] and Foreshadow-NG [292], we systematically analyzed all the remaining bits in the page tables as well as other exceptions that can be exploited with Meltdown-type attacks as well as predictors exploited with Spectre-type attacks. This uncovered 6 new attacks.

The paper "A Systematic Evaluation of Transient Execution Attacks and Defenses" was published at *USENIX Security Symposium* 2019 in collaboration with Claudio Canella, Jo Van Bulck, Michael Schwarz, Benjamin

von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss.

ConTExT. In order to fully defend against transient-execution attacks, we investigated if a software-hardware co-design is possible that would protect sensitive data but keep the performance gain introduced by speculative execution. Thus, we proposed ConTExT, which makes the hardware aware of secrets by annotating them in code and passing this information on to the lower levels, and forcing the hardware not to use secrets when speculating.

The paper "ConTExT: A Generic Approach for Mitigating Spectre" was published at *NDSS* 2020 in collaboration with Michael Schwarz, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss.

ZombieLoad. Within our Meltdown research [177], we discovered a variant that allowed leaking values not only from the cache but also from the Line Fill Buffer (LFB). With ZombieLoad, we continued in that direction to leak data from the load buffers. We were able to leak currently processed data of a thread running in parallel or previously processed data on the same core. With those techniques, we can leak loads and stores from load ports, the line fill buffer, and the store buffer. With one particular variant, called TSX Asynchronous Abort (TAA), we demonstrated that these attacks even work on CPUs that already have hardware mitigations against the other variants. We further demonstrate that the initial microcode mitigations by Intel were insufficient, and L1D Eviction Sampling (L1DES) still allows to leak sensitive data.

The paper "ZombieLoad: Cross-Privilege-Boundary Data Sampling" was published at *ACM CCS* 2019 in collaboration with Michael Schwarz, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss.

Fallout. Another optimization technique to resolve data hazards is store-to-load forwarding, where the CPU passes data from previous stores onto subsequent loads. With Fallout, we show that we can exploit this behavior and trick the CPU into forwarding previous stores from the victim to the attacker as long as they are in the store buffer.

The paper "Fallout: Leaking Data on Meltdown-resistant CPUs" was published at *ACM CCS* 2019 in collaboration with Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom.

LVI. With Load Value Injection (LVI), we turn around previous transient-execution attacks directly extracting data [46, 177, 251, 293]. Rather than directly leaking the data from the victim, we inject data into the victim to hijack transient execution leaking sensitive information. We show that LVI is harder to mitigate as it requires expensive software patches to prevent transient execution after every load operation.

The paper "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection" was published at *IEEE S&P* 2020 in collaboration with Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens.

Medusa. With Medusa, we investigated whether a fuzzing-based approach allows us to find new Meltdown subvariants. All existing variants of Meltdown have been found with the manual effort by experts in the field and fixed, either using microcode updates or for upcoming CPU microarchitectures directly in silicon. However, certain subvariants like TAA [122] or L1DES [124] demonstrated that not all paths had been covered in hardware and, thus, were applicable to even the latest generation of CPUs. We introduce Transynther, an automatic approach that allows reproducing, analyzing, and classifying existing variants and also generating new variants and regression testing. Based on our findings, we identify a new variant called Medusa that leaks data from write-combining memory operations. Furthermore, Transynther synthesized a variant of Fallout [46] that worked even on the most recent Intel Ice Lake microarchitecture [196].

The paper "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis" was published at *USENIX Security Symposium* 2020 in collaboration with Daniel Moghimi, Berk Sunar, and Michael Schwarz.

1.3. Outline

The remainder of this thesis is structured as follows. Chapter 2 provides background on the architecture, microarchitecture, and memory organization of a modern CPU. Furthermore, it introduces side-channel attacks, fault attacks, and microarchitectural attacks. Chapter 3 gives an overview of the state of the art in software-based microarchitectural side-channel attacks, fault attacks, and power side-channel attacks. Chapter 4 concludes this work and gives an outlook on ongoing and future research.

2
Background

In this chapter, we provide the necessary background for this thesis. In Section 2.1, we explain architecture and microarchitecture and describe trusted-execution environments. In Section 2.2, we explain how memory is organized in modern processors, covering the concepts of virtual memory and caches. We discuss the basic concepts of side-channel attacks and fault attacks in Section 2.3.

2.1. Architecture and Microarchitecture

In this section, we discuss the importance of abstraction to deal with complexity in computing by means of computer architecture and microarchitecture. We briefly introduce the concepts of pipelined processors and superscalar techniques of modern processors.

2.1.1. Instruction-Set Architecture

By using abstraction layers, one needs to take care only of the interface or specification of each layer. Thus, higher levels do not necessarily need to know the details of lower levels. In computer science, abstraction layers play an important role in managing the complexity of modern systems, and, therefore, they do not only exist between hardware and software but basically everywhere.

Levels of Abstraction. Every abstraction level in a modern computer system, as illustrated in Figure 2.1, is implemented on top of another level and utilizes the well-defined functions of the lower level. While the implementation of each layer is not interested in the upper layers, it hides the unnecessary details of the lower layers. Starting with atoms, this allows us to build complex applications.

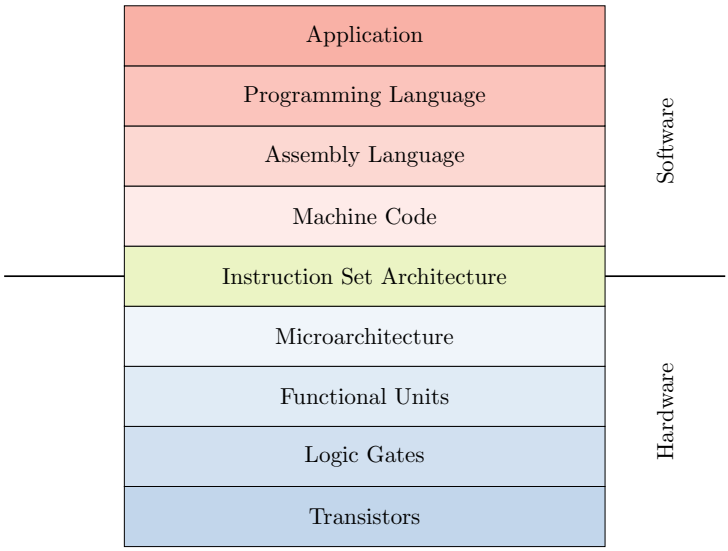


Figure 2.1.: Abstraction layers enable building complex processors from the bottom up. The ISA connects the hardware to the software level.

Transistors built from silicon atoms are used for amplification and to build switches. By combining these switches, boolean logic gates (such as `or`, `xor`, `and` or `not`) can be built. Using multiple gates, functional blocks, such as latches, flip-flops, and registers, are built. On the next level, they can be chained together to build more complex logical functions that perform computations, like arithmetical logical units (ALU). To design an entire processor, multiple complex processing elements are built and connected together to enable complex computations. This includes register files, different buffers, and execution units that together represent the microarchitecture that fetches instructions from memory, decodes and executes them, and stores the results back to memory. The computer architecture (or Instruction Set Architecture (ISA)) is an abstract model of the machine and connects hardware and software. A compiler uses the ISA to translate a high-level programming language to the machine code the processor can process. This allows applications to be executed on the CPU. Typically, an operating system manages how the system is used and enables user space applications to run on top of it that operate on data we provide.

Instruction-Set Architecture (ISA). The ISA, or computer architecture, is an abstract model of the computer. It defines not only the instruction a processor can execute as well as their behavior but also registers, data types, and the memory model. With x86, A64, POWER, RISC-V, or SPARC, there are many different instruction sets that are typically classified in their complexity. A complex-instruction-set computer (CISC), like x86, supports many special (and hence, complex) instructions. On the other hand, a reduced-instruction-set computer (RISC), like A64 or POWER, has a smaller set of instructions and, thus, requires more instructions to perform tasks than some of the complex instructions would cover.

Usually, a RISC instruction set uses fixed-length instructions. This means, as an example with the ARM A64 instruction set, that all instructions are 32 bits in length [24]. In contrast, CISC instruction may have instructions of varying lengths. For instance, instructions on x86 can be between 1 and 15 bytes long [123]. Hence, parsing an instruction stream is more complex, as the actual length of the next instruction can only be determined by parsing byte by byte.

With computer architecture, one usually refers to the Instruction-Set Architecture (ISA) of a processor and, thus, the processor's interface as the programmer sees it. From the hardware perspective, the ISA serves as the design specification for the microarchitecture. This enables different microarchitectures (from different companies) for the same ISA, allowing to run applications compiled for one instruction set to run on all of these microarchitectures. However, based on the actual implementation of the microarchitecture, the program's execution can vary in side effects, such as performance or energy consumption.

While we focus on the CPU microarchitecture in Section 2.1.2, the distinguishing view between the architecture and microarchitecture applies to abstraction levels in general. An architecture always describes the interface to the underlying microarchitecture. For example, the operating system provides a clear interface for applications via specified system calls and signals. Different versions of an operating system can be seen as different microarchitectures as well; however, if an application adheres to the interface, it can be executed on different implementations of the operating system. For instance, while the internal workings – including different optimizations – of the operating systems, can change, an application can still be executed correctly if the guarantees of the interfaces are kept.

2.1.2. CPU Microarchitecture

The microarchitecture of a processor is an implementation of an architecture. While the ISA serves as a reference for software developers, it serves as the interface specification for CPU designers that the microarchitecture must meet.

In this section, we briefly discuss some fundamental techniques of modern microarchitectures. We outline pipelined designs, as well as superscalar techniques like out-of-order execution. In addition, we examine performance optimization techniques like hardware-based speculation.

For a more in-depth compendium of these topics, we refer the reader to *Computer Architecture: A Quantitative Approach* [103] and *Modern Processor Design: Fundamentals of Superscalar Processors* [268].

2.1.2.1. Pipelining

Pipelining allows increasing the throughput of a system, *i.e.*, the number of tasks a system can perform per time unit. Thus, it plays an important role in the design of modern processors to increase their performance.

With pipelining, the execution of a task is split into multiple stages (subtasks) with buffers between each stage. This allows a new task to start as soon as the previous task has completed the first stage of the pipeline.

Pipelines are not only used to perform arithmetic operations (*arithmetic pipelines*), e.g., floating-point multiplications, but also to pipeline the instructions the processor should perform in each instruction cycle (*instruction pipelines*). A simple RISC pipeline, as illustrated in Figure 2.2, consists of the following 5 stages:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Write-back (WB)

In the first stage (IF), the next instruction to be executed is fetched. Then, the instruction is decoded (ID) to determine the work the instruction should perform. In the execute stage (EX), the actual computation is performed. If the operation needs to access memory, this is handled in the memory stage (MEM). Finally, in the write-back stage (WB), the

computed results are written to the register file.

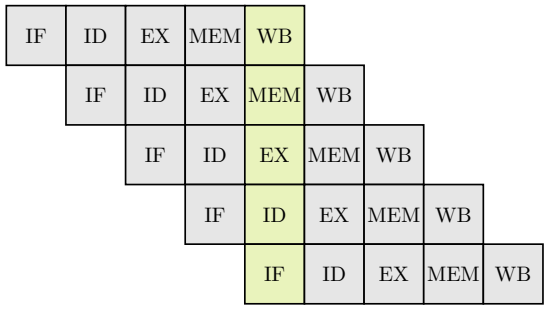


Figure 2.2.: A RISC Pipeline consisting of 5 stages Instruction fetch (IF), Instruction decode (ID), Execute (ED), Memory Access (MEM), and Write Back (WB).

In an n-stage pipeline, n different instructions can be processed at the same time. If these instructions do not depend on each other, instructions can continue through the pipeline without any stalls. However, when there are dependencies between instructions within the pipeline, they have to be detected and resolved. Conditional branches cause pipeline hazards (problems that occur when the next instruction cannot execute in the following clock) as the outcome of a conditional branch is unknown until the execution stage, but its result is already required in the instruction fetch stage to load the next instruction. A simple solution is to assume that the branch falls through and continue fetching the next instruction speculatively but stalling at the decode stage. If the predictions turn out to be correct and the branch is not taken, only a single cycle is lost. However, if the branch has been taken and the assumption was incorrect, the new target has to be fetched and, thus, two cycles have been lost. While this simple prediction allows gaining performance over stalling completely, there are more sophisticated prediction techniques discussed in Section 2.1.2.3.

Hazards cannot only occur on the instruction flow but also on the data flow. If there are two pipeline stages in the pipeline that can simultaneously access the same variable, different data hazards can occur. For example, one instruction could need the value that is computed by a previous instruction in the pipeline. With this read-after-write (RAW) hazard, the following instruction must be prevented from entering the pipeline. Thus, the pipeline must stall until the preceding operation has finished. However, the performance of resolving pipeline hazards can be

improved. Using forwarding paths, the result of the preceding operation could be forwarded to the depending operation, reducing the penalty that would be necessary if the pipeline needed to be stalled completely.

Precise and non-precise exceptions. In every pipeline stage, different exceptions can interrupt the execution of several instructions. For instance, a page fault or a memory-protection violation could occur in the instruction fetch stage. In the decoding stage, the decoder could fail to decode the instruction, or the instruction cannot be executed with the current privilege level. In the execution stage, the actual computation could raise an exception, *i.e.*, if a number is divided by 0. Similar to the instruction-fetch stage, page faults or protection violations could occur in the memory access stage.

If an exception occurs, the pipeline must squash all following instructions and should let all preceding instructions complete. If these conditions are met and, thus, the machine state aligns with the sequential execution of the instruction stream, exceptions are called precise. If, however, these conditions are not met, e.g., preceding instructions are not completed if an exception occurs, the exceptions are called non-precise. Typically, modern processors support precise exceptions.

Furthermore, it can happen that multiple exceptions occur simultaneously in more than one pipeline stage. If an exception occurs, the pipeline started to perform operations that should not have been executed and, thus, has to make sure that these have no architectural effect.

To handle exceptions on the architectural level, the operating system must provide entry points for *exception and interrupt handlers* in a so-called interrupt descriptor table (IDT). If an exception is architecturally raised (when the instruction would be committed and, thus, the architectural state be updated), the processor calls the procedure for the exception from the IDT [118].

2.1.2.2. Superscalar Execution

With superscalar execution, pipelines are parallelized so that multiple instructions can be processed in every cycle. They are further diversified by using multiple different functional units in the execution stage and by implementing dynamic pipelines; out-of-order execution allows maximizing the utilization of all execution units.

Parallel and Diversified Pipelines. A simple approach to increase the performance of a pipelined CPU is to create multiple copies of the same pipeline to run in parallel. However, this does not only increase the complexity but also the required hardware resources significantly.

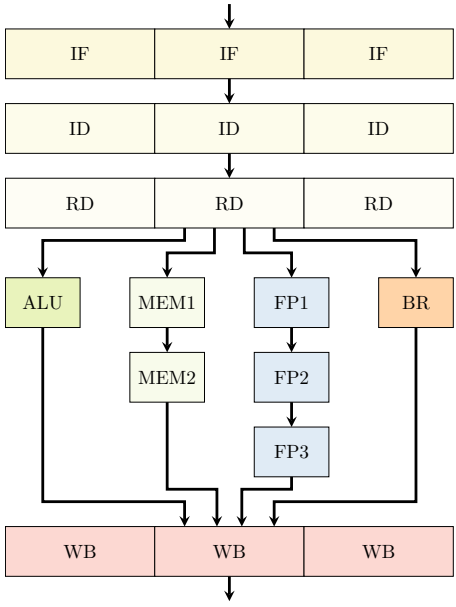


Figure 2.3.: In the execution stage of diversified pipelines, multiple functional units are implemented [268].

With diversified pipelines, however, not every stage in the pipeline is duplicated. Instead, in the execution stage, diversified execution pipelines using multiple functional units are implemented, e.g., for ALU, memory, branch (BR), or floating-point operations, as illustrated in Figure 2.3. Thus, individual pipelines customized for instruction types can be built, and, thus, the latency for each instruction type can be minimal. For instance, the ALU pipeline finishes with a single cycle and does not have to wait for the completion of all floating-point stages.

Dynamic Pipelines. For instructions to complete in the same order as defined by the instruction stream of the program in a scalar pipeline, the preceding pipeline stages must also be stalled whenever an instruction must be held back in a pipeline buffer. For parallel pipelines, multi-entry buffers are necessary. If the entire multi-entry buffer is, like a single-entry buffer, either stalled or clocked in each cycle, multiple instructions that

in theory require no stalling would also be blocked, inducing unnecessary performance loss. To minimize the latency, subsequent instructions must be able to bypass stalled instructions. Hence, with dynamic pipelines, instructions can be executed out-of-order, *i.e.*, the order instructions are executed deviates from the order defined by the instruction stream. Thus, instructions are executed as soon as their operands are ready. Using complex multi-entry reordering buffers, as shown in Figure 2.4, the instructions are first brought in order to the dispatch buffer, where they are dispatched to the execution units as soon as their operands are ready. With the reorder buffer, while finishing out-of-order, they are reordered back in order to be written back and committed to the architectural state.

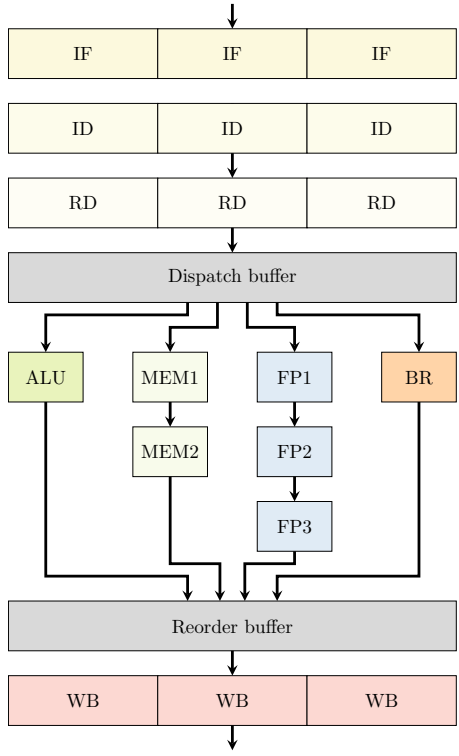


Figure 2.4.: Dynamic Pipelines use complex buffers to enable out-of-order execution [268].

In 1967, Tomasulo [286] developed an algorithm that enabled dynamic scheduling of instructions to allow out-of-order execution. It introduces a reservation station, a common data bus, and register tags. A reservation station is a buffer attached to each execution unit that holds the currently

executed instructions of that instruction type. This allows dispatching instructions to execution units as long as there are free reservation stations, even if the execution unit is busy executing a different instruction and even if their operands are not available yet. Thus, instructions can wait in the reservation station until the operands become ready. The common data bus connects the outputs of the execution units to the reservation stations, allowing the instructions waiting to load their required operands from the bus. Similarly, the destination registers of these instructions are also updated. To implement this, register renaming is used. The register contains either a real value or a tag that indicates which execution unit will compute the value. This tag, along with its corresponding result, is broadcast on the common data bus by the producing execution unit, and instructions waiting for their operands are monitoring for their tags on the bus. If an instruction is dispatched, the tag of its output register is stored with the id of the execution unit. Likewise, if a following instruction with the same output register is dispatched, the tag will be updated with the id of the execution unit of the new instruction. Thus, the tag will always contain the id of the execution unit of the last instruction. Thus, only the latest instruction can therefore update the value of the register, resolving write-after-write hazards. However, with Tomasulo's design alone, precise exceptions cannot be handled as the register file is not updated by all instructions. For precise exceptions, modern CPU designs use a *reorder buffer*, as discussed in Section 2.1.2.4.

Out-of-order designs allow to execute operations *speculatively* to the extent that instructions can be processed before the processor is certain that they are actually needed and their results committed to the architectural state. While the out-of-order design enables the execution of instructions that lie completely outside of the program's order, note that Tomasulo's algorithm, in his nature, does not perform anything speculatively. We briefly discuss performance optimizations using speculative execution techniques like branch predictors in Section 2.1.2.3 and further building blocks in Section 2.1.2.4.

2.1.2.3. Speculative Execution

The instruction streams that a processor is executing are usually not only linear but rather contain branches diverting the control flow. To improve the performance, different prediction mechanisms are implemented that try to make an educated guess, which instruction will be executed next.

Besides branch prediction and branch-target prediction, explained below, other prediction-based techniques increasing the performance have been proposed and deployed on modern systems. Examples include data prefetching [142], way prediction [227], or value speculation [218].

Branch Prediction and Branch-Target Prediction. With branch predictors, the processor tries to determine in which direction a branch is taken before its condition has been evaluated. Instructions on the predicted path can be executed in advance, and their results used if the prediction turns out to be correct. However, if the prediction was incorrect, the wrongly computed results are discarded, and execution is continued from the correct path.

There are different ways to predict a conditional branch: Using *static branch prediction* [103], the outcome is predicted based on the instruction itself. For instance, every conditional branch is predicted to be taken. With dynamic branch prediction [54], information is gathered during the run-time enabling an educated guess for the outcome. For instance, one-level branch prediction uses a 1-bit or 2-bit counter to record the last outcome of a branch [268]. Two-level adaptive predictors [337] use the branch history to look up the saturating counter in a pattern-history table (PHT). With local branch predictors, a separate history is kept for each conditional jump instruction, while global predictors share the history of all conditional jumps. With neural branch prediction [143, 284, 304], ideas from machine learning have been integrated into CPU architectures [18, 245].

If the direction of a branch is known (or predicted), the address of the next instruction must be determined. The same applies to unconditional branches that are always taken. For conditional branches, if the branch is not taken, the next instruction is simply the one following the current branch instruction. If a branch is taken, however, the address can either be given (direct branch) or computed at runtime (indirect branch). With branch-target prediction, the processor tries to predict the target address of the branch. The branch target buffer (BTB) is used to store the best-predicted target address for a branch instruction and is consulted to predict the target of the next instruction that should be fetched.

Not only calls into a function can be predicted, but also the return address can be predicted using a return-stack buffer (RSB) [106, 144]. If a function is called, the return address is pushed onto the RSB. When

returning from the function, the last entry of the RSB serves as a target prediction from where the processor should continue executing. However, as the return stack has a limited size, an overflow can overwrite the oldest return address, yielding a misprediction. Intel's return-stack buffers are implemented as a circular array, while AMD's RSBs have overflow and underflow checking [322].

Uzelac and Milenkovic [290] presented experiments to reverse-engineer the structures of branch predictors on the Intel Pentium M. Kocher et al. [156] reverse-engineered the BTB from Intel's Haswell microarchitecture. Bhattacharya et al. [34] aimed to reverse-engineer the prediction model of Intel's branch predictors. Wong micro-benchmarked the return-stack buffer on different microarchitectures [322]. We discuss many other works that exploit these predictors to leak sensitive data in Section 3.1.

2.1.2.4. Modern Microarchitecture

Most superscalar processors consist of a front-end, fetching and dispatching instructions in order, an out-of-order execution engine, and a back end that retires instructions in order, as well as the memory subsystem. In this section, we briefly discuss state-of-the-art microarchitectures based on Intel's Skylake core architecture [255], as illustrated in Figure 2.5.

Front End. In the front-end, x86 instructions are fetched from memory, decoded to micro-operations (μOPs) and sent to the execution engine. The *branch predictors* decide which instructions are fetched as macro-ops from the L1 instruction cache (L1I) and sent to the pre-decode buffer. As x86 instructions are complex and of varying length, their boundaries are marked in the pre-decode phase. In the *instruction queue*, the pre-decoded instructions are further optimized by macro-op fusion, where two macro-ops are combined into a single complex operation.

Using multiple, simple and complex *decoders* macro-operations are decoded into fixed-length μOPs. While simple decoders can only emit single μOPs, the complex decoder can decode up to four μOPs. For instructions that require more than 4 μOPs and cannot be decoded by the complex decoder, the *microcode sequencer* (MS ROM) is queried. While it emits the required μOPs, the decoders are inactive.

Finally, the decoded μOPs are send to the *Allocation Queue* that serves as an interface between the front end and the execution engine. There,

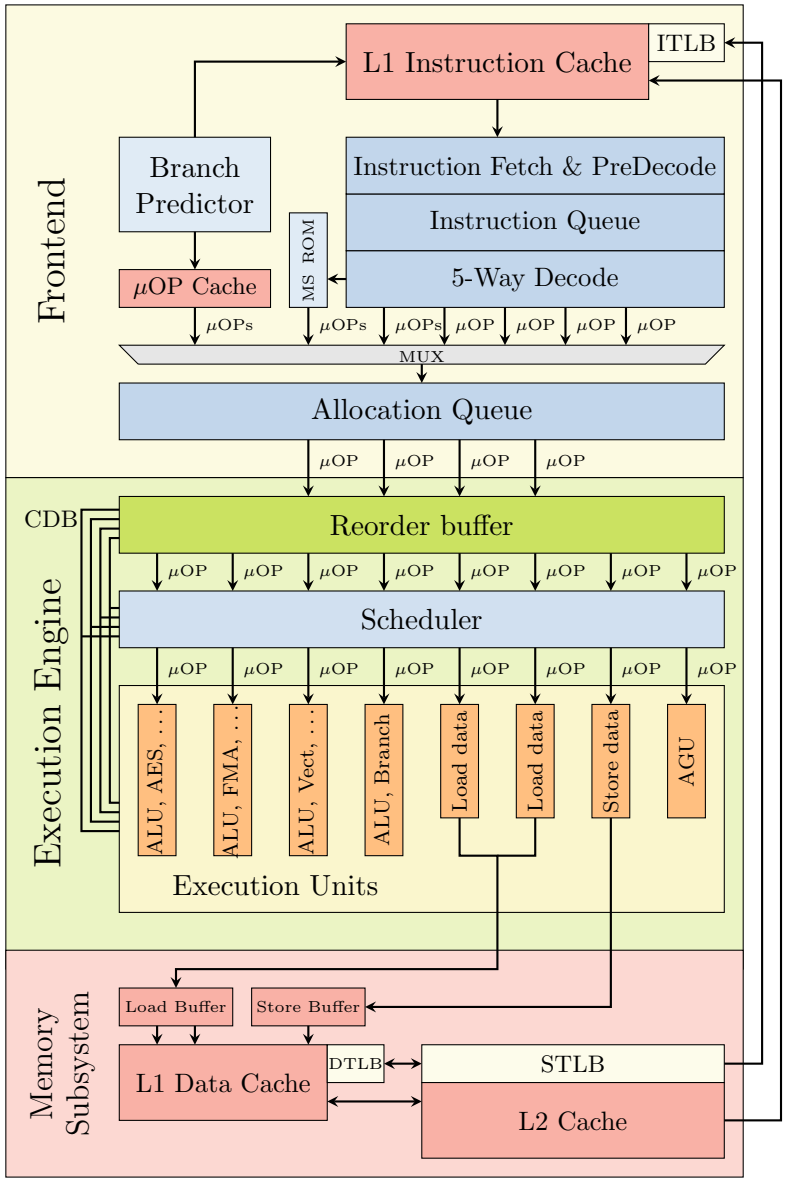


Figure 2.5.: A simplified view of the Intel Skylake microarchitecture [255].

additional optimizations take place: A loop stream detector detects loops and sends the μOPs directly to the execution engine bypassing the rest of the front end [117]. Furthermore, micro-fusion allows fusing multiple μOPs from the same instruction into a single complex instruction [117].

To improve the performance and avoid repeatedly decoding the same instructions, a *μOP cache* is used to store the μOPs of already decoded instructions. If there is a hit in the μOP cache, the μOPs can be directly emitted to the allocation queue avoiding the pre-decode and decoding stage.

Execution Core. The superscalar execution core of the Skylake microarchitecture can process instructions out of order. The instructions arrive in the reorder buffer (ROB) from the allocation queue of the front end. The reorder buffer holds μOPs in different stages of completion. First, if necessary, additional resources for the μOPs are allocated, e.g., entries in the load or store buffer. The source and destination registers are mapped to physical registers, and the register alias table performs register renaming to enable out-of-order execution as described in Section 2.1.2.

The *reservation station* (or scheduler) queues μOPs until their operands are ready, schedules and dispatches them to the corresponding execution units. The reservation station is connected (via ports) to execution units performing different instructions like ALU operations, floating-point operations, multiplications, memory loads, and stores. For instance, port 0 is capable of integer and vector arithmetic operations as well as AES instructions. Port 2 and 3 are for memory loads, while port 7 is for memory stores. If an execution unit finishes executing one instruction, it broadcasts the result on the common data bus such that depending instructions can fetch the results they require to be executed directly.

The reorder buffer contains all *in-flight* instructions that have been dispatched but not yet retired, including instructions that are waiting for their operands and instructions that have been completed out of order but are not yet committed. The reorder buffer also makes sure that instructions retire in order and, thus, makes it appear that the instructions have been executed as specified by the instruction stream. With Tomasulo's algorithm, the register file is updated once an instruction finishes, and subsequent instructions can hence find the result there. However, with the reorder buffer, the register file is not updated until the instruction retires, thus, allowing the core to execute instructions speculatively.

Furthermore, precise exceptions are implemented with the reorder buffer. If an exception occurs, the entry of the instruction that triggered the exception is marked in the reorder buffer. While completing instructions, the reorder buffer checks whether the instruction has been marked and,

thus, is not allowed to complete. Therefore, instructions that preceded the faulting instruction can still retire in order. The results of instructions that have been executed out of order that follow the faulting instructions are discarded.

Memory Subsystem. The memory subsystem is responsible for load and store instructions as well as the ordering of memory accesses. While we discuss the memory organization, including caches, virtual memory, and DRAM, in more depth in Section 2.2, we give a short overview of the steps performed by the core for memory operations as well as the involved hardware blocks.

The execution of a memory instruction consists of three steps: the memory address generation, the address translation and the actual memory access. There are different modes how memory addresses can be computed. If an instruction uses an absolute memory address, it uses it directly. Otherwise, using a base and index register as well as a scale and displacement value, based on the form $base + index \cdot scale + displacement$, different combinations are possible [118]. Thus, in the first step, the actual memory address is computed based on the provided values for the used mode. If virtual memory (see Section 2.2.2) is used, the computed virtual address has to be translated to a physical address that is used to access the physical memory. Using so-called page tables, the address translation is performed. Furthermore, translations are cached in translation-lookaside buffers (TLB), speeding up subsequent accesses to the same virtual addresses.

After computing and translating the virtual address, the actual memory access can be performed. The data is retrieved from the data cache and stored in the renamed register or the reorder buffer. Note that the architectural register is only updated if the load instruction completes in the reorder buffer [268]. If the data is not cached in the data cache, a cache miss occurs, and the data is requested from the main memory, inducing a longer delay for the instruction to finish. Store instructions are handled differently than load instructions, as store instructions finish as soon as the address has been translated [268]. However, the data is only stored in memory iff the store completes in the reorder buffer. Thus, speculatively but erroneously executed instructions cannot affect the actual memory contents.

Similar to registers, dependencies between two load or store instructions

can exist if they refer to the same memory location. With the execution core we described above, also memory instructions can be performed out of order; thus, special care has to be taken to tackle possible data hazards. Using *load bypassing*, loads can be executed earlier than preceding stores if there is no data dependence between the stores and load. However, if there is a read-after-write (RAW) dependency between a store and a load, using *load forwarding*, the load retrieves its data directly from the store instead of accessing the memory. The memory-order buffer (MOB), consisting of a load buffer and store buffer, is used to handle the memory requests. The load buffer queues loads that could not complete when they were dispatched by the reservation station. It also snoops for stores of other cores against completed loads to maintain memory ordering. The store buffer, on the other hand, queues all stores before they are dispatched to memory in order (when they are no longer speculative). If a load is issued, the store buffer is checked for a potential address match, e.g., using the lower 12 bits, such that data from a previous store operation could be directly forwarded to the load operation [105, 107].

2.1.2.5. Trusted-Execution Environments

Trusted-Execution Environments (TEE) are a secure and isolated environments in CPUs that protect code and data from an otherwise untrusted system. Different CPU manufacturers provide different TEEs: ARM's TrustZone [25] is probably the most widely-used TEE as it is available in most mobile phones. With the Armv9-A architecture, ARM introduced the Confidential Compute Architecture (CCA) [23] that builds upon TrustZone to provide additional execution environments, so-called realms. With Secure Encrypted Virtualization [146] and Secure Nested Paging [17], AMD provides different technologies to isolate virtual machines from the hypervisor. With Secure Execution [112], IBM enables running protected Linux virtual machines in the cloud.

With Software Guard Extension (SGX), Intel provides an x86 instruction-set extension protecting applications in so-called enclaves [59]. By encrypting the memory, neither applications nor the operating system can inspect the memory used by an enclave. In addition, this successfully protects against cold-boot attacks [121]. Furthermore, local and remote attestation ensures the integrity of an enclave before it is loaded and executed. To communicate with enclaves, applications can use ecalls to call functions provided by the enclave. Using ocalls, an enclave can request

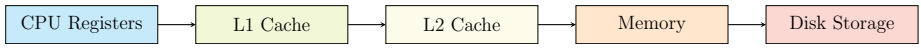


Figure 2.6.: Memory hierarchy shows different locations where data can reside. The closer to the CPU core, the faster the access time, e.g., loading data from one of the cache levels is faster than accessing the data from the main memory.

functionality provided by the operating system, as an enclave itself is restricted in its functionality. For instance, an enclave can not perform any I/O operations or system calls by itself.

We discuss microarchitectural attacks targeting trusted-execution environments in Chapter 3.

2.2. Memory Organization

The performance of modern processors does not only depend on the clock frequency but is also influenced by the latency of instructions and, especially, the interaction with other components such as the main memory or hard drives. In this section, we discuss the memory hierarchy, caches, TLBs, and DRAM. We further explain the concept of virtual memory and ASLR.

2.2.1. Memory Hierarchy

To serve memory requests as fast as possible, a memory hierarchy as illustrated in Figure 2.6 is deployed. Frequently used data is buffered in multiple layers that differ in capacity and speed. Closer to the core, the memory becomes smaller but faster using different storage technologies:

Disk Storage
Usually, the slowest but largest memory is the disk storage, e.g., a hard disk, at the end of the hierarchy. While the capacity of hard disks and SSDs reaches multiple terabytes, the latency of a hard disk is around 4 ms [167] while the latency of an SSD can be as low as 250 µs [246].

Memory
Typically, DRAM is used for the main memory that is discussed

in greater detail in Section 2.2.5. Modern systems deploy multiple gigabytes of memory with latencies around 60 ns [168].

Caches
Modern microarchitectures deploy multiple levels of cache, typically using SRAM. The lower the level, the smaller it's capacity and lower its latency. For instance, the L1 cache has a latency of 4 cycles. The L2 needs 12 cycles, and the L3 around 30 cycles to serve the requests [117]. We discuss caches in greater detail in Section 2.2.3.

CPU Registers
Register files hold the general-purpose registers and are built using multi-ported SRAM. Typically, moving a register to another register has a latency of 1 cycle [2]. However, using move elimination [182], the register renaming performs the move and, thus, no latency is induced.

Usually, programs tend to reuse the same memory locations over time repeatedly as well as memory locations that are close to each other, forming two aspects of locality [272]: With *temporal locality*, if one memory location is accessed at a particular time, then it is likely that it will be referenced again soon. With *spatial locality*, if one memory location is accessed at a particular time, then it is likely that nearby locations will be used soon as well. These two principles play an important role in the design of caches to hold frequently used data close to the core.

2.2.2. Virtual Memory

For memory isolation, processors support *virtual memory* as an abstraction layer for the physical memory of the system. Instead of assigning the physical address space to each process, each process has its own *virtual address space* organized in *pages*. The physical memory is divided into fixed-size continuous blocks, so-called *page frames*. Using multi-level page-translation tables, the operating system maps virtual pages to page frames. When accessing a virtual address of a process, the CPU resolves the mapping to operate on the actual physical address. The root of a translation table for a process is stored in a dedicated register, e.g., the `CR3` register on x86 architectures. When switching processes, the operating system switches to the address space of the next process by updating this register.

On modern processors, these translation tables typically have 4 levels,

as illustrated in Figure 2.7. However, with the recent Ice Lake microarchitecture, Intel supports another level and, thus, 5-level paging. Every paging structure has a size of 4 kB and consists of 512 entries of each 8 B. Therefore, with 4 levels, 48 bit of the virtual address are used to index the different page table levels (with 5 levels, 57 bits are used).

The `CR3` register points to the top-most page map level 4 (PML4). Bits 39 to 47 of the virtual address are used to select one of the 512 entries of the PML4, pointing to the page directory pointer table (PDPT). Bits 30 to 38 of the address determine which of the PDPT entries is selected. A PDPT entry either defines a 1 GB region of physical memory (a 1 GB page) or points to a page directory (PD). Bits 21 to 29 of the address define the PD index. Similarly, a PD entry either maps to a 2 MB region of physical memory (a 2 MB or huge page) or maps again to a so-called page table (PT). Bits 12 to 20 select the page table entry that then maps to a 4 kB page of physical memory. With 5-level paging, an additional table, page map level 5 (PML5), is used.

In addition to the page frame number, the entries in the translation tables store additional properties of the virtual address. For instance, these properties can define if the process can write to the page or if the process can execute code from the page. Furthermore, for *memory protection*, a bit defines if this address can be accessed from user and kernel space, or only from kernel space only. This allows the operating system to map pages into the virtual address space of the user space process that can only be accessed by the kernel.

2.2.3. Caches

As discussed in Section 2.2.1, a memory hierarchy is employed to overcome the latency of memory accesses. Data that has been used recently is stored in multiple layers of fast and small memories, so-called caches. Thus, access to data in any of the caches is significantly faster than those to the main memory.

In this section, we discuss different cache organizations and cache properties.

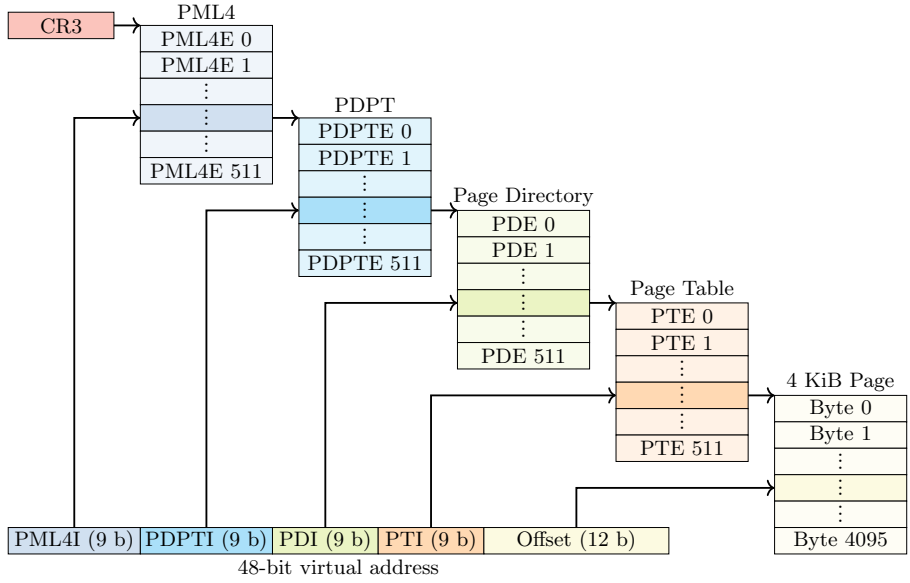


Figure 2.7.: Address translation on x86 processors [83]. Different bits of the virtual address select the entry of paging tables.

2.2.3.1. Cache Organization

Due to their comparably small sizes, caches only hold a subset of the contents of the main memory. A cache stores data divided into blocks, so-called cache lines, that are typically 64 B in size. If a cache line for the requested memory address can be served from the cache, it is called a *cache hit*. If it has to be requested from the main memory, it is called a *cache miss*.

As illustrated in Figure 2.8, typically, there are multiple levels of caches in a system. The cache closest to the core is called the first-level cache (L1) and is typically split into an instruction cache and a data cache. The next level, the L2 cache, is a unified cache and stores both instructions and data. Both the L1 and L2 cache are private to each core. The last level cache (LLC), the L3 cache, is shared among all cores and is slower but larger than the lower levels. Furthermore, the LLC is split into so-called *slices* where each core can access one slice directly and the others over a bus.

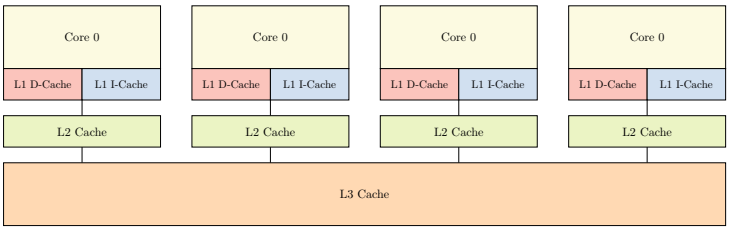


Figure 2.8.: There are typically multiple levels of caches in a system. In this example, each core has a private L1 instruction and data cache as well as an L2 cache. The last-level cache (L3) is shared among all cores.

Cache Inclusion Policy. When multiple levels of caches are used, some design decisions have to be made which cache levels hold copies of the data, e.g., if a copy of the cache line is kept by one or multiple caches [272]: A higher-level cache is called *inclusive* with regard to the lower-level cache if all cache lines from the lower-level cache are also stored in the higher-level cache. Caches are called *exclusive* to each other if a cache line can only be kept in one of the cache levels. If a cache is neither *inclusive* nor *exclusive*, it is called *non-inclusive*. Most modern Intel CPUs have inclusive last-level caches [201], and AMD CPUs have a non-inclusive or exclusive last-level caches [128]. ARM CPUs have non-inclusive [81, 174] or exclusive [19].

Cache Designs. There are different ways to implement a cache, with the simplest one being a *direct-mapped* cache. In a direct-mapped cache, each cache line in the main memory maps to a single location within the cache.

Figure 2.9 illustrates a direct-mapped cache with 8 locations and cache lines of 64 B in size. Thus, the cache controller uses the lowest 6 bits of the 32-bit address to select a word within the cache line. To map the address to one of the 8 possible locations, 3 bits of the address are used. The remaining 23 bits will be stored as a tag value such that the requested memory location can be uniquely identified and checked against.

If an address is looked up in the cache, the index bits of the address are extracted and used to select a row in the cache. Then the tag value of the address is compared to the tag value stored in the selected row. Only if there is a match and the entry in the cache is marked valid it is a cache

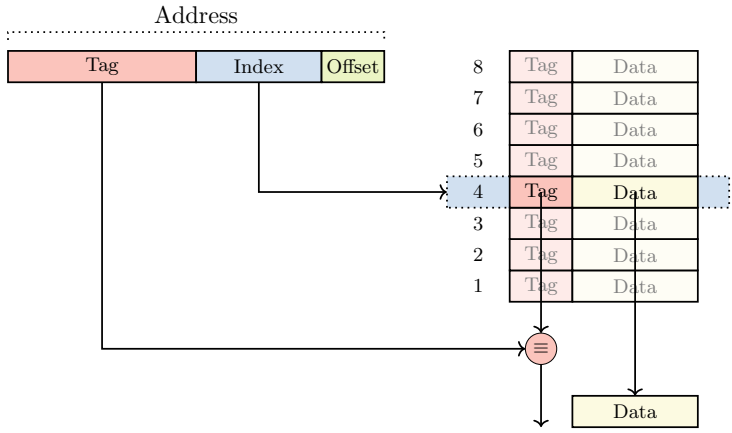


Figure 2.9.: In a *direct-mapped cache*, each cache line maps to a single location within the cache.

hit, and the relevant word can be extracted using the offset bits of the address. If, however, either the valid bit is not set or the tag does not match, a cache miss occurs, and the data has to be requested from the memory or higher cache levels.

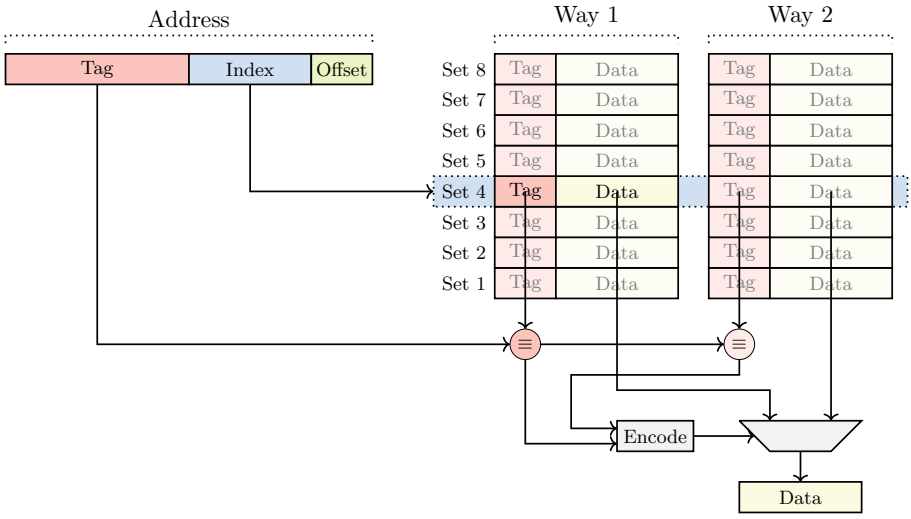


Figure 2.10.: In an n-way associative cache, each cache line maps into a cache set with n ways where a cache line can be stored. The illustration shows a 2-way cache.

The cache is called *fully associative* if an address can be stored at any

entry in the cache. If caches are organized in sets of cache lines, they are called *set-associative* caches. Each set contains multiple *ways* where a cache line can be stored. If a cache line can be stored in any of n places in the cache, the cache is n-way associative, as illustrated in Figure 2.10. When accessing a cache line, each line stored in each way for a set must be compared to the requested address to select to correct one. Addresses that map to the same cache set are called *congruent addresses*. Congruent addresses compete for cache lines within the same set, and a cache-replacement policy needs to decide which cache line will be replaced.

Cache-replacement Policies. If all entries in a cache set are used, and a cache miss occurs, a cache line must be evicted from the cache set to make room for a new entry [1]. The used heuristic that determines which cache line to evict is called *cache-replacement policy*. For instance, a least-recently-used (LRU) policy would replace the cache line that has been the least recently used. A pseudo-random replacement policy will select a cache entry and evict it based on a pseudo-random number generator. Fine-grained replacement policies [135] learn from the behavior of previous cache lines to be more effective. While the lower-level caches of Intel processors use pseudo LRU (PLRU) replacement algorithms [1, 302], the L3 cache uses an adaptive policy [321] using adaptive insertion [234] and Re-reference Interval Prediction (RRIP) [137]. Vila et al. [302] reverse-engineered undocumented replacement algorithms for the L2 and L3 cache on Intel CPUs.

Virtual and Physical Tags and Indexes. The address used to index the cache can be a virtual address or a physical address. Thus, a CPU cache can either be *virtually indexed* or *physically indexed*. Virtually-indexed caches are faster as they do not require an address translation before the lookup can be performed. However, depending on the design of the cache, situations can occur where the same physical address is stored in different cache lines. Furthermore, the tag that is stored along with the cache line can also be based on the virtual or physical address, allowing different combinations with their advantages and disadvantages:

With *VIVT* (virtually indexed, virtually tagged), the virtual address is used for both the index and the tag, which improves performance since no address translation is needed. However, the virtual tag is not unique, and

shared memory may be held more than once in the cache. In addition, the entries need to be either tagged with a process identifier, e.g., PCID, or flushed on each context switch as they are not unique across processes. For *PIPT* (physically indexed, physically tagged) caches, the physical address is used for both the index and the tag. This method is slower since the virtual address has to be looked up in the translation-lookaside buffer (TLB). However, shared memory is only held once in the cache. With *PIVT* (physically indexed, virtually tagged), the physical address is used for the index, and the virtual address is used for the tag. This combination has no benefit since the address needs to be translated, the virtual tag is not unique, and shared memory can still be held more than once in the cache. *VIPT* (virtually indexed, physically tagged) caches use the virtual address for the index and the physical address for the tag. The advantage of this combination compared to PIPT is the lower latency since the index can be looked up in parallel to the TLB translation. However, the tag cannot be compared until the physical address is available. Most Intel CPUs have a VIPT L1 cache [118], where PIPT caches are used for caches of higher levels.

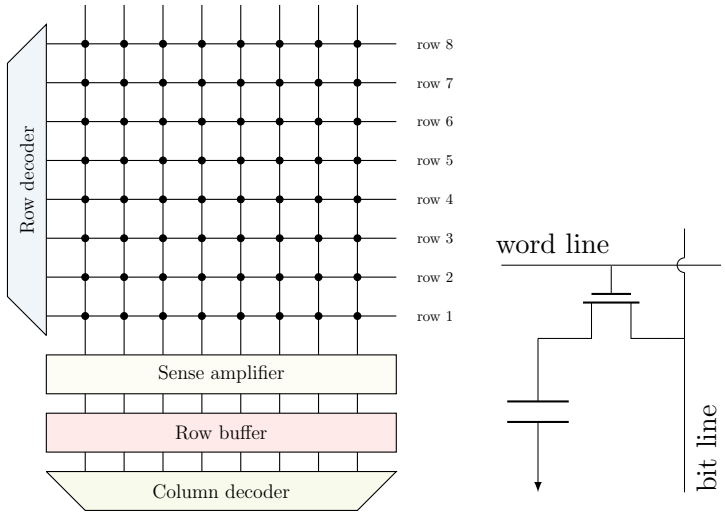
2.2.4. Translation Lookaside Buffer (TLB)

With virtual memory, the processor maps virtual addresses to physical addresses by walking the page tables as discussed in Section 2.2.2. For every load or store operation, the page tables that are stored in physical memory have to be accessed to determine the physical address and the page attributes. To improve the performance, dedicated caches called translation-lookaside buffers (TLB) are used to cache page-table entries [193]. For every memory operation, the TLB is queried for the entry of a virtual address, and only if the translation has not been cached before, the *page-miss handler* (PMH) performs a page-table walk. During each context switch, the TLB must be invalidated in order to avoid old and incorrect translations. With global pages (defined by the global bit in the page table) or when using process identifiers (PCID) [118], not all entries cached in the TLB need to be invalidated.

2.2.5. DRAM

The main memory of a computer system is typically built using DRAM chips. DRAM chips are manufactured in different configurations, varying in their capacity, bandwidth, and latency.

DRAM Organization. To achieve a high degree of parallelism, DRAM is organized in a hierarchy of channels, DIMMs, ranks, bank groups, and banks. The memory controller translates physical addresses to channel, DIMM, rank, bank group, bank, row, and column addresses to address the correct data [268]. A single DRAM bank consists of a two-dimensional array of cells, as illustrated in Figure 2.11a. Each single DRAM cell is made out of a capacitor and an access transistor, as shown in Figure 2.11b. The charge state of the capacitor, either charged or discharged, represents a binary data value. Each cell in the grid is connected to the neighbor cell with a wire forming a horizontal *word line* and a vertical *bit line*. If a word line of a row is raised to a high voltage, all access transistors in that row are activated, thus, connecting all capacitors to their respective bit line. By doing that, the charge representing the data of the row is transferred to the so-called *row buffer*.



(a) Rows of cells (b) Single cell

Figure 2.11.: A DRAM chip uses an array of DRAM cells that each consist of a transistor and capacitor. The word line and bit lines are used to select the cells that will be loaded into the row buffer.

DRAM Accessing. To access data in a memory bank, the desired row needs to be opened at first by raising the corresponding word line, as illustrated in Figure 2.11. By that, the row is connected to all bit lines,

the data is sensed by the sense amplifier and transferred into the row buffer. Then the data in the row buffer is accessed and modified by reading or writing in the row buffer. If data from a different row but in the same bank needs to be accessed, the current row needs to be closed by lowering the corresponding word line, and the row buffer is cleared.

However, the charge stored in the capacitor of a DRAM cell is not persistent because its charge can disperse over time. This means that after some time, data is lost. To avoid this, the cell's charge must be refreshed by fully charging or discharging it. DRAM specifications require that all cells in a rank are refreshed within a certain amount of time, the so-called refresh rate [141].

With Rowhammer, this effect can be exploited to induce bit flips in DRAM. We discuss state of the art in software-based fault attacks targeting DRAM in Section 3.3.

2.2.6. Address Space Layout Randomization

To successfully exploit memory corruption bugs, the knowledge of addresses of specific data or functions is often required. A lightweight and probabilistic defense against such attacks is Address Space Layout Randomization (ASLR) [215]. With ASLR, the positions in the virtual address space of the executable, stack, heap, and libraries of a process are randomized at the start of the process. To defeat the randomization, an attacker must successfully guess the address of the data required for the attack. With increasing entropy, a successful guess becomes more unlikely.

Kernel ASLR (KASLR) applies ASLR in the operating system [64]. By randomizing the locations of kernel code, data, and drivers on every boot, this mitigation strategy impedes the exploitation of kernel bugs. With fine-grained KASLR (FGKASLR) [3], Intel proposed a new implementation for the Linux kernel. Instead of rearranging the entire kernel binary, the kernel code is randomized at boot time on a per-function level granularity.

However, various side-channel attacks [49, 90, 110, 140, 158, 175] have been demonstrated in the past that either reduce the entropy of KASLR or allow to break it entirely. We discuss them in more detail in Section 3.1.

2.3. Side-Channel Attacks and Fault Attacks

In this section, we briefly discuss the basics of side-channel attacks and fault attacks. We outline software-based microarchitectural attacks but discuss the more recent attacks and state of the art in Chapter 3.

2.3.1. Side-Channel Attacks

Side-channel attacks exploit the indirect information leakage of software implementations or hardware devices to reconstruct sensitive data [149]. The execution time of an algorithm, the power consumption, or the electromagnetic emanation of a device provides additional information that an adversary can exploit. Furthermore, the additional information must not necessarily be directly revealed by the device itself. Hall et al. [100] observed the reactions of users using the device as an additional information source. However, an information source that is published on purpose could also serve as a side channel [275]. For instance, the memory footprint [138] or operating system statistics [274] allow deducing sensitive information, e.g., visited websites. With a side channel attack an adversary can only leak meta data, e.g., the device uses more or less energy, but not the data directly. However, in a successful attack the adversary can infer the corresponding data from the obtained meta data with high probability.

Side-channel attacks can be categorized in different ways. While *active* attacks actively influence the behavior of the device, *passive* attacks only monitor the side channel in a passive manner. For instance, an adversary that actively delayers a chip to obtain his measurements is *invasive*. On the contrary, a *non-invasive* adversary obtains his measurements without modifying the system under attack. In a *local* attack, the adversary requires direct physical access to the device, e.g., to obtain power measurements. With *vicinity* side-channel attacks [275], an attacker has to be in the vicinity of the targeted device, e.g., to monitor the Wi-Fi signals [15]. However, side-channel attacks can also be conducted *remotely*, e.g., by measuring response times over the network [264]. However, side-channel attacks that are software-only can also be seen as remote attacks if an adversary can run his exploit on the victim machine, either by an installed application or within sandboxed JavaScript in the browser.

In attacks against the implementation of cryptographic algorithms, an attacker can make use of the property that the measurements can be re-

peated as often as necessary, allowing to average out measurement noise. However, side-channel attacks in many scenarios cannot profit from that. For example, when monitoring one-time events, like user input, the attacker typically has only one chance of observing the event as the attacker cannot force the victim to type the passphrase as often as required. For user input, the attacker can detect the exact points in time where the user pressed a key on the keyboard and, thus, compute the inter-keystroke timings. These timings allow an attacker to compute, to some extent, which words a user typed by correlating the inter-keystroke timings with the distances between characters and the likelihood that they belong together. As side-channel attacks only use metadata, such attacks are very hard to mount without profiling the victim first as users tend to type in different ways.

Covert Channels. In computing, a communication channel transmits information from one or several senders to one or several receivers. Following the definition of Lampson [162], a *covert channel* transfers information despite not being a *legitimate channel*, *i.e.*, a channel that has been explicitly designed to be used to transmit information. In a covert channel, both the sender and receiver are controlled by the adversary to transmit data, enabling communication where both the receiver and sender are not allowed to communicate using legitimate channels or want to hide their communication. With a *side channel*, however, the victim serves as the sender of the channel. At the same time, the adversary controls only the receiving end of the channel to observe and infer the data. Note that the actual channel used can be the same for a covert channel and a side channel, *i.e.*, the energy consumption or the CPU cache.

As these terms are often used interchangeably, Intel introduced *incidental channels* allowing to clearly distinguish between the usage of side channels and covert channels [127]. All channels that are not legitimate channels are incidental channels. An incidental channel is used as a covert channel if the adversary controls both the sender and receiver. An incidental channel is used as a side channel if the adversary can not control the sender but only the receiver.

2.3.2. Fault Attacks

Fault attacks deliberately manipulate the device under attack to induce errors within its computations by bringing the device to the edge of or

outside its specified working conditions, e.g., violating voltage or temperature constraints. A fault attack consists of two steps: *fault injection* and *fault exploitation*.

The first step is to inject a fault at the right moment of time (when the device executes the targeted operation) to corrupt or skip the computation. This can be achieved by bringing the device outside of its normal operating conditions. A fault can be induced by manipulating the clock (clock glitching) or voltage (voltage glitching), exposing the device to a high temperature, or using a laser beam. The second step is the actual exploitation of the fault and depends on the induced fault. For instance, an induced bit flip could flip the permission bit in a page table, allowing an unprivileged user to access arbitrary memory [299]. A prominent class of fault attacks is differential fault analysis (DFA), where faults are induced in cryptographic implementations to reveal their inner states [38].

2.3.3. Software-based Microarchitectural Attacks

Software-based Microarchitectural attacks target the actual microarchitectural implementation of an ISA specification (see Section 2.1.2) by crafting side-channel and fault attacks purely in software. We discuss the state-of-the-art of software-based microarchitectural attacks in more depth in Chapter 3.

Software-based Microarchitectural Side-Channel Attacks. Using *microarchitectural side-channel attacks*, observable side effects on the microarchitectural level are exploited to leak sensitive information. Many microarchitectural side-channel attacks target the cache of the processor, or other microarchitectural elements such as the TLB [80] or branch predictors [68]. With caches, the timing difference introduced by data being cached or not allows an attacker to leak sensitive information. Alongside *Prime+Probe* [287], *Evict+Time* [287], or *Flush+Reload* [335], many different attack variants have been published. These variants allow to attack cryptographic algorithms [287, 334, 335], to monitor user behavior [94, 174, 220], spy on virtual machines [114, 131], or attack ASLR [79] and kernel ASLR (KASLR) [89, 110, 140, 320].

Transient-Execution Attacks. More recent attacks on other microarchitectural elements, *transient-execution attacks*, bypass the most funda-

mental security guarantees of modern processors. These attacks exploit the transient execution of instructions that, while executed, are not committed to the architectural state by the processor. However, during their execution, they leave microarchitectural state changes that an adversary can observe. Meltdown-type attacks [26, 50, 119, 126, 155, 177, 276, 292, 293] exploit transient execution of instructions before a fault is handled, while Spectre-type attacks [50, 52, 109, 155–157, 183] exploit the transient execution of instructions caused by mispredictions. These attacks allow the execution of instructions operating with the actual data processed by the CPU. They typically use the cache as a covert channel to transmit the data from a microarchitectural state to an architectural state. With transient-execution attacks, a new research field emerged in the field of microarchitectural attacks.

Software-based Microarchitectural Fault Attacks. While traditional fault attacks (see Section 2.3.2) require physical access to the device to induce faults from the outside, *software-based microarchitectural fault attacks* aim to induce faults on the microarchitectural level from software only.

The Rowhammer bug was the first software-based microarchitectural fault attack, inducing bit flips in DRAM cells. Targeting the DRAM microarchitecture, bit flips can be induced by repeatedly accessing DRAM cells in a high frequency [153]. By inducing bit flips in memory regions that are not controlled by the adversary, powerful attacks have been demonstrated: from privilege escalation [88, 266, 299, 327] to fault attacks on cryptographic primitives [35, 238], to denial-of-service attacks [88, 139].

Karimi et al. maliciously simulated aging effects in CPU cores to degrade the processor's performance by 10.91 % in 4 weeks [147]. Tang et al. [282] use software-controlled frequency-induced faults to break ARM TrustZone. Plundervolt [202], Voltjockey [233], and V0LTpwn [150] reduce the voltage of modern processor cores to induce faults within computations of trusted-execution environments, compromising their security guarantees.

3

State of the Art

In this chapter, we discuss state-of-the-art software-based microarchitectural attacks and defenses. We discuss software-based microarchitectural side-channel attacks in Section 3.1, transient-execution attacks in Section 3.2, and fault attacks in Section 3.3. In Section 3.4, we will discuss software-based power side-channel attacks.

3.1. Software-based Microarchitectural Side-Channel Attacks

In this section, we provide an overview of side-channel attacks targeting the CPU microarchitecture by executing software. We discuss observable variations in the behavior of different microarchitectural elements often introduced by performance optimizations that allow leaking sensitive information processed on the machine. Numerous works already provided an exhaustive overview and systematized the landscape of microarchitectural attacks over the past years [7, 33, 74, 83, 181, 205, 275, 280, 338, 340]. However, the field is growing at a fast pace, and new attacks or defenses emerge almost daily. In this work, we discuss the current state-of-the-art and how we extended it.

Microarchitectural Side-Channel Attack. Typically, a microarchitectural side-channel attack consists of 3 phases:

1. **Preparation**: The adversary sets the microarchitectural element to a known state.
2. **Schedule**: The victim performs an event changing the state of the microarchitectural element.
3. **Observation**: The adversary detects the state change of the microarchitectural element.

In the first phase, the *preparation phase*, the adversary sets the microarchitectural element to a known state. The microarchitectural element needs to be shared between the victim and adversary. As an example, the adversary ensures that the cache is filled with its own data. In the second phase, the victim is *scheduled*, *i.e.*, the victim process can perform an operation that changes the state of the microarchitectural element. Note that the second phase is not necessarily executed sequentially or triggered by the attacker. Many of the described attacks can be mounted in parallel to the victim and, thus, a victim action might not be triggered all the time. In the *observation* step, the adversary detects the state change that the victim has performed; typically, by observing timing differences. The meta-information of detecting the state change enables the attacker to infer that the victim has performed a certain operation.

Timing Measurement. For timing-based side-channel attacks, in most cases, a high-resolution timer is used to distinguish subtle timing differences. While most attacks utilize the `rdtsc` instruction on x86 CPUs, on other architectures such timers are not available to unprivileged user space or not exposed to adversaries running in unprivileged environments. Hence, in addition to `rdtsc`, alternative timers have been studied in the literature: We explored a dedicated counting thread, unprivileged access to hardware performance counters or the use of POSIX functions as alternative measurement sources on ARM-based devices [174]. Schwarz et al. [263] explored alternative timing sources in JavaScript where native counters are not accessible.

Shared Elements. Microarchitectural side-channel attacks target microarchitectural elements that are accessible (directly or indirectly) for the adversary and the victim. For instance, some microarchitectural elements are private to each CPU core, e.g., the L1 data and instruction cache. To be accessible to both the attacker and the victim, they have to run on the same physical core such that both have access to the L1 caches. Some of the elements are statically partitioned between hardware threads, and, thus, the attacker and victim even need to run on the same logical core. On the other hand, some elements are shared between all cores, e.g., the last-level cache, or among all CPUs, e.g., the main memory.

In addition, the attacker typically targets victims from a higher privilege domain (the operating system, hypervisor, or sandbox) or processes that

are isolated from the attacker (other user-space applications, trusted-execution environments). However, in virtualized (virtual machines), sandboxed (web browser), or other restricted environments (Intel SGX, TrustZone), access to these resources can be limited or restricted in different ways.

In the remainder of this section, we discuss different attacks on different microarchitectural elements, namely the cache, the TLB, various prediction mechanisms, the main memory, and exception handling.

3.1.1. Cache Attacks

The most prominent microarchitectural element exploited for side-channel attacks is the cache. In this section, we discuss different types of cache attacks that have been used to establish a covert communication channel, attack cryptographic implementations, and to monitor user activity.

Evict+Time. With *Evict+Time*, Osvik et al. [211] generalized the first cache attacks [32, 219] where an adversary manipulates the cache state and observes timing differences in the execution time of the algorithm under attack. By evicting a cache set, the adversary replaces the cache lines in the cache set by loading its own data ensuring that victim-controlled cache lines are not cached anymore. First, the adversary triggers an event, e.g., an encryption, and, thus, ensures that the memory blocks by the victim are cached. The attack consists of 2 steps:

1. **Evict**: By evicting congruent addresses, a specific cache set is primed (loaded with the adversaries data).
2. **Time**: The adversary measures the execution time of the event again.

If the adversary measures a difference in the execution time, the victim accessed cache lines that map to the primed cache set as accesses to these cache lines will yield cache misses and, therefore, have to be loaded from main memory. If, on the other hand, the adversary does not observe an increase in the execution time, the victim did not access cache lines within the primed set.

Usually *Evict+Time* attacks are quite susceptible to noise and, therefore, require a large number of measurements. *Evict+Time* has been used to

attack cryptographic implementations [32, 136, 174, 192, 211, 219, 287], ASLR [79], and KASLR [110].

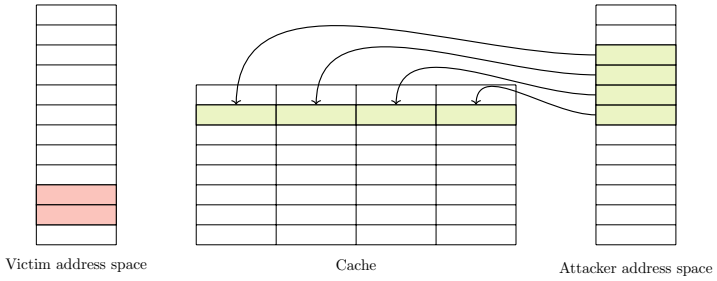
Prime+Probe. With *Prime+Probe* [32, 211, 219], the adversary measures how long it takes to refill a specific cache set in contrast to *Evict+Time* where the execution time of the entire algorithm is measured. The attack consists of 3 steps:

1. **Prime**: The adversary primes a cache set by filling the specific set with attacker-controlled memory locations (Figure 3.1a).
2. *Schedule*: The victim is scheduled, e.g., performs an encryption (Figure 3.1b).
3. **Probe**: The adversary measures how long it takes to reload the addresses used in the Prime step (Figure 3.1c).

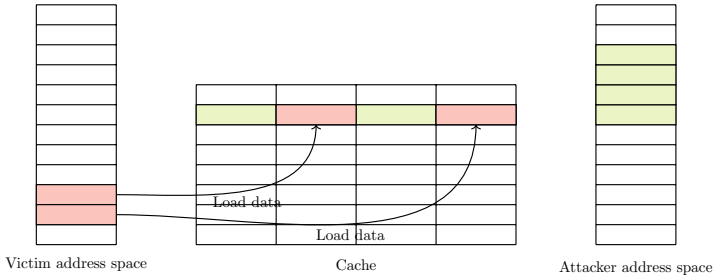
Likewise to *Evict+Time*, *Prime+Probe* attacks work on a cache-set granularity. They can be quite noisy and, thus, require a higher repetition of measurements. *Prime+Probe* is often mounted in parallel to the victim's execution and, thus, the victim is not directly scheduled in the second step. However, depending on the attack, the signal can be amplified by monitoring multiple cache sets at the same time. With *Multi-Prime+Probe* [261], it is possible to detect one-time events such as keystrokes.

Prime+Probe has been used to attack cryptographic implementation targeting the L1 data and L1 instruction caches [4–6, 8, 9, 40, 44, 206, 211, 219, 343] and also the last-level cache [61, 97, 115, 129, 148, 180, 188, 189, 239, 242, 310]. As the last-level cache is physically indexed, physical address information is typically required to find addresses for the eviction. Oren et al. [210] used *Prime+Probe* in sandboxed JavaScript to recover information of other processes, users, and virtual machines. Lipp et al. [174] mounted *Prime+Probe* on mobile devices, Maurice et al. [190] established a covert channel in the cloud, and Schwarz et al. [261] monitored inter-keystroke timings targeting kernel drivers. Several attacks target Intel SGX enclaves [42, 77, 195, 258] or mount their attack from an enclave [265]. With NetCAT [160], Kurth et al. demonstrated the first network-based *Prime+Probe* attack using Direct Cache Access (DCA). While most attacks have been demonstrated on inclusive caches, Yan et al. [332] use *Prime+Probe* on non-inclusive caches targeting cache directories.

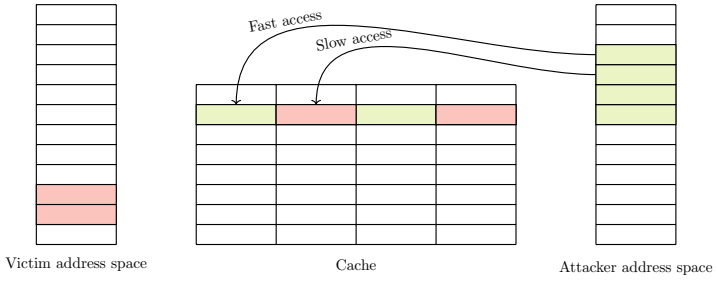
However, the *Prime+Probe* attack technique is not limited to CPU caches



(a) `Prime`



(b) *Schedule*



(c) `Probe`

Figure 3.1.: In a *Prime+Probe* attack, the adversary first primes the cache set by loading its own data. If the victim accesses addresses of the same cache set, the adversary measures a higher timing the probe phase.

and has been demonstrated against all kinds of buffers. Pessl et al. [220] targeted the DRAM row buffer to monitor keystrokes. Bhattacharya et al. [35] combined *Prime+Probe* with Rowhammer in a cryptographic attack. Using *Prime+Probe* on branch predictors, Aciicmez et al. [10] attacked

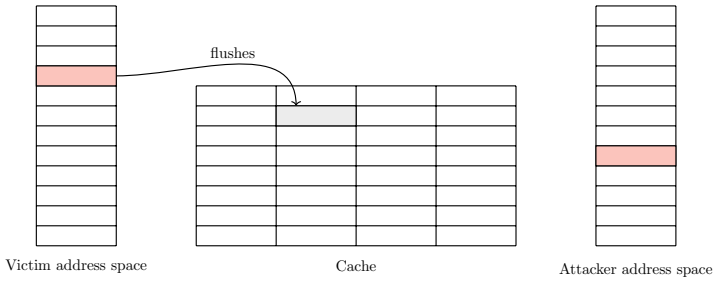
RSA, while Evtyushkin established a covert channel [66] and demonstrated an attack against KASLR [67].

Flush+Reload. While *Evict+Time* and *Prime+Probe* attacks typically target an entire cache set, and, thus, work on a cache-set granularity, the *Flush+Reload* attack targets a single cache line. While Gullasch et al. [96] described the first flush-based cache attack, Yarom and Falkner [335] demonstrated with *Flush+Reload* an attack that measures how long it takes to reload a single cache line after it has been flushed from the cache. The attack consists of 3 phases:

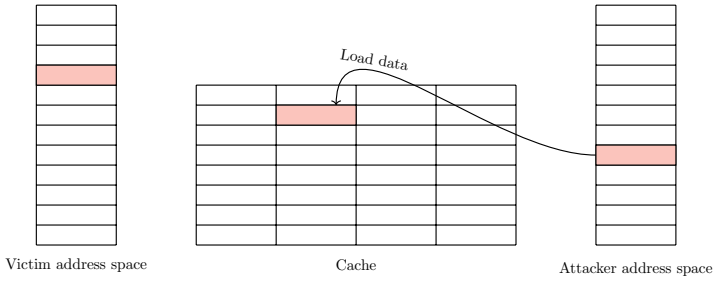
1. **Flush**: The adversary invalidates a single cache line from the cache using a dedicated instruction (Figure 3.2a), e.g., `clflush`.
2. *Schedule*: The victim is scheduled and performs an event (Figure 3.2b), e.g., performs an encryption.
3. **Reload**: The adversary measures how long it takes to reload the address used in the Flush step (Figure 3.2c).

The idea is that if the victim performed an access to the invalidated address, the victim loads said address into the cache. Thus, the adversary observes a cache hit rather than a cache miss that would occur if the victim did not access the address. Typically, this attack is performed in a loop to monitor accesses by the victim continuously. While *Flush+Reload* experiences very little noise, it has, however, 2 limitations. First, the memory used for the attack needs to be shared between the victim process and the adversary process. This can be achieved by targeting the executable of the victim or a shared system library. Second, a dedicated cache-line invalidation instruction must be accessible to the unprivileged attacker. On x86, an adversary can use `clflush` or `clflushopt`; on ARMv8-based devices, `dc civac`. However, such instructions can be disabled on ARM for unprivileged access, and while they are not available on all architectures, they are also not exposed to restricted environments such as JavaScript.

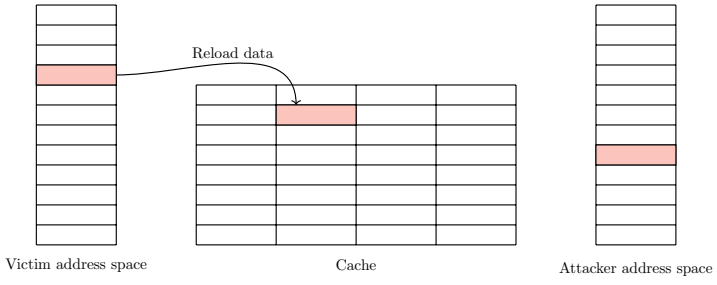
Flush+Reload has been demonstrated in attacks against cryptographic implementations [16, 31, 82, 96, 98, 114, 131–133, 174, 225, 335, 342], to monitor user interaction [94, 174, 199, 309, 341] like user input or as a covert communication channel [156, 174, 177].



(a) `Flush`



(b) *Schedule*



(c) `Reload`

Figure 3.2.: With *Flush+Reload*, the adversary invalidates a cache line shared with the victim. If the victim accesses the cache line, the adversary observes a low access time in the reload step.

Evict+Reload. Flush instructions are not available to unprivileged user space on certain architectures at all, e.g., ARMv7, or not exposed to the user in sandboxed environments or interpreted languages. Therefore, *Evict+Reload* [94, 174] describes a special variant of *Flush+Reload* that replaces the flush step, that invalidates the targeted cache line, with eviction. The targeted cache line is evicted when the replacement policy decides to remove the cache line from the cache and replaces it with other data. In the most simple form, the adversary loads random data into the

cache that the targeted cache line will be evicted with high probability. By using efficient eviction, adversaries can mount similar attacks on these devices [174], in JavaScript [79, 156, 251] or remotely [264]. Song and Liu [271] and Vila et al. [303] present techniques to efficiently find eviction sets.

Flush+Flush. With *Flush+Flush*, Gruss et al. [92] demonstrated a variant of *Flush+Reload* that performs no direct memory accesses. In contrast, it exploits timing differences in the `clflush` instructions to distinguish cache hits from misses. *Flush+Flush* has been used to break cryptographic algorithms [92, 297], monitor user input [92] and to establish covert communication [92, 174]

Prime+Abort. *Prime+Abort* [63, 86] is a variant of *Prime+Probe* that replaces the timing of the probe step with the abort semantics of Intel TSX. When the victim process evicts a cache line that has been loaded within the TSX transaction by the adversary, the transaction is aborted. This allows the adversary to detect memory accesses by the victim without requiring a high-resolution timer.

Other Cache Attacks. With Reload+Refresh, Briongos et al. [43] abused cache-replacement policies of the LLC. Cui and Cheng [60] use timing differences caused by dirty cache lines to establish a covert channel. Wan et al. [308] exploit timing differences in the congestion of the CPU mesh interconnect. Maurice et al. [188], Irazoqui et al. [130], and Yarom et al. [336] reverse-engineered the complex addressing functions of Intel's last-level caches. Xiong et al. [328] target the LRU replacement policy to establish a covert channel.

With Takeaway (Chapter 5), we advance the state of the art by exploiting timing differences induced by cache-way predictors as a side channel.

3.1.2. TLB Attacks

The translation-lookaside buffer (TLB) stores the page translations from virtual to physical memory to speed up subsequent accesses to the same address. The timing difference, whether an address has already been translated and cached in the TLB or if the translation is uncached, can be exploited in multiple ways.

These timing differences have been exploited to break KASLR in various ways [46, 90, 110, 140, 158, 256]. For instance, there is a timing difference in the `prefetch` instruction if an address is cached in the TLB or not [90].

Gras et al. [80] exploited the TLB to leak fine-grained information about victim processes to leak EdDSA keys. Deng et al. [62] modeled different timing-based TLB attacks. Schwarz et al. [256] further exploited the interaction between the store buffer and the TLB to break ASLR from JavaScript and to monitor the control flow of the kernel.

3.1.3. Predictors

Predictors play an important role in optimizing the performance in modern microarchitectures using speculative execution. Besides branch predictors that try to predict the control flow of programs, other predictors try to predict the data flow.

Branch Predictors. Aciicmez et al. [10] exploited timing differences that are caused by mispredicted branches to recover cryptographic keys. Bulygin [45] mounted a side channel on the return-stack buffer to leak cryptographic keys and to establish a covert channel between virtual machines. Evtyushkin et al. [66] established a *Prime+Probe* covert channel on branch predictors, followed by breaking KASLR [67] using BTB collisions. With BranchScope, Evtyushkin et al. [68] presented a fine-grained attack on the directional branch predictor to attack Intel SGX. Similarly, Lee et al. [166] presented a BTB side-channel attack inferring the control flow of SGX enclaves. With BlueThunder, Huo et al. [111] demonstrated a PHT-based side-channel attack against Intel SGX.

Way Predictors. With Takeaway (see Chapter 5), we advance the state of the art by exploiting way predictors. AMD uses cache way predictors to predict in which cache way a certain address is located. We reverse-engineered the cache way predictors resulting in two attack techniques: With Collide+Probe, an attacker can monitor a victim's memory accesses without knowledge of physical addresses or shared memory when time-sharing a logical core. With Load+Reload, highly accurate memory-access traces of victims on the same physical core can be obtained. We demonstrated these techniques to establish a covert channel, break ASLR

and KASLR, recover AES keys, and to exfiltrate secret data from the kernel.

Store-to-load Forwarding. With *store-to-load forwarding*, a CPU forwards the data from a previous store operation to a subsequent load operation. As discussed in Section 2.1.2, the data is typically forwarded after the processor asserts that the address of the store and load match. Schwarz et al. [256] exploit this optimization in combination with the TLB to break KASLR. On Intel CPUs, loads are checked against previous stores, but instead of comparing the full address, only the lower 12 bits have to match before the load is reissued [117]. This effect is also known as *4K aliasing*. In addition, partial address checks based on the physical address are performed as well [134]. Sullivan et al. [278] exploited this effect to establish a high-performance covert channel. With MemJam, Moghimi et al. [194] inject false dependencies of memory read-after-write hazards that slow down the victim's access to specific memory blocks. Using a similar technique as *Evict+Time*, they successfully recovered secret keys from a variety of constant-time cryptographic implementations. With Spoiler, Islam et al. [134] exploited the dependency resolution logic to learn about physical address mappings accelerating *Prime+Probe* attacks and Rowhammer attacks.

In contrast to store-to-load forwarding, AMD's predictive store forwarding does not wait for the address calculation to complete [11]. By learning the relationship between loads and stores within the same context over time, a predictor speculatively performs store-to-load forwarding.

Prefetcher. Bhattacharya et al. [36] explored the influence of hardware prefetchers in cache timing attacks. Shin et al. [269] exploit stride prefetchers to monitor access patterns, allowing to leak cryptographic keys. With PAPP, Wang et al. [310] reverse-engineered hardware prefetchers and their replacement policies to build a prefetcher-aware *Prime+Probe* attack. Rohan et al. [243] reverse-engineered the L2 hardware prefetcher and used it to establish a covert channel between two processes running on the same physical core.

3.1.4. DRAM

The main memory, usually implemented as DRAM, plays another important role in the memory hierarchy (see Section 2.2.1). By design, DRAM modules (see Section 2.2.5) contain row buffers that are necessary to read from and write to rows in the chip. Memory requests to a row that have already been opened are faster as they are still buffered in the row buffer. With DRAMA, Pessl et al. [220] exploit these timing differences caused by row hits and row conflicts in DRAM modules to not only reverse-engineer the mapping functions of the memory controller but also monitor user activity and establish a covert channel. Furthermore, Barenghi et al. [28], Wang et al. [312] , and Helm et al. [102] discuss alternative approaches to reverse-engineer the mapping functions. Schwarz et al. [263] exploited this measurable effect in JavaScript to build a DRAM-based covert channel. We used this timing difference to automatically classify the page policies actively used by the memory controller [179]. Furthermore, side-channel attacks targeting the DRAM have been mounted on SGX enclaves [313] or by monitoring the bandwidth [314].

3.1.5. Exceptions and Interrupts

Another category of microarchitectural side-channel attacks targets exceptions and interrupt handling that disrupts the execution of instructions. While exceptions are unexpected events occurring within the execution of instructions, e.g., page faults, interrupts are events triggered from outside the program, e.g., hardware break points, user input, or timers. Whenever an exception or interrupt occurs, it has to be handled by the operating system before the execution of the program can continue.

In this section, we discuss several works that exploit exceptions and interrupts to deduce sensitive information. Suzaki et al. [279], Owens et al. [212], Xiao et al. [325], Bosman [41] and Gruss et al. [85], Harnik [101] used page faults caused by memory deduplication to detect running processes and fingerprinting. Xu et al. [330], Van Bulck et al. [296], Xiao et al. [326] and Weiser et al. [317] induce page faults in controlled-channel attacks against Intel SGX enclaves. With Nemesis, Van Bulck et al. [295] measured the latency of timer interrupts to infer instruction timings of Intel SGX enclaves.

With KeyDrown [261], we monitored processor interrupts to infer inter-keystroke timings. We demonstrated the applicability of the attack in

JavaScript on desktop and mobile devices (see Chapter 8), allowing us to observe user activity for touch and tap events, *i.e.*, when a user enters the unlock pin code of the phone.

3.1.6. Other Microarchitectural Side-Channel Attacks

In addition to the discussed categories of microarchitectural side-channel attacks, further research on other microarchitectural elements has been conducted. Sibert et al. described several possible side-channel attacks on the Intel 80x86 processor already in 1995, e.g., on the FPU, the cache, and TLB. Wang [315] identified various attack vectors in processors in 2006: From the contention of functional units in SMT processors to speculation-based covered channels. Aldaya et al. [13] exploited port contention to recover cryptographic keys. Andrysco et al. [20] identified timing differences in the floating-point instructions depending on their operands. Evtyushkin et al. [65] targeted the random number generators on processors to establish a covert channel across CPU cores and across virtual machines. Paccagnella et al. [213] target the CPU on-chip ring interconnect. Uhsadel et al. [289] and Bhattacharya and Mukhopadhyay [35] exploit hardware performance counters to conduct side-channel attacks. Jang et al. [140] exploit transactional memory to derandomize the kernel address-space layout (KASLR). Canella et al. [49] exploit a side channel introduced by incomplete Meltdown mitigations to break KASLR. Weber et al. [316] present a fuzzing-based framework to automatically discover microarchitectural side channels.

3.2. Transient-Execution Attacks

Transient-execution attacks exploit the microarchitectural side effects of instructions that are executed transiently, *i.e.*, instructions that are executed by the processor but their results are never committed to the architectural state. While microarchitectural side-channel attacks have been limited to leak only metadata about the execution of a program, *i.e.*, executed instructions or data accesses, transient-execution attacks extract victim data directly. In this section, we discuss transient-execution attacks in general, their different variants, and defenses.

Basic Idea. With speculative execution (see Section 2.1.2.3), processors try to predict which instructions should be executed next, leading to the

transient execution of these instructions if the prediction was false. With out-of-order execution (see Section 2.1.2), operations following an instruction triggering an exception can also be executed transiently. While transient instructions are never committed to the architectural state, their execution alone can cause observable side effects in the microarchitectural domain. Using microarchitectural side-channel techniques, these side effects can be made visible in the architectural domain.

Transient-execution attacks describe a new class of microarchitectural attacks that access secret data in the transient domain and use a microarchitectural covert channel to transmit secret data to the architectural domain. Thus, transient-execution attacks consist of two main parts: the access to secret data and the covert communication channel. The access to secret data could be a location in the victim's memory that the victim instruction stream would usually not access, e.g., an out-of-bounds memory access. Furthermore, as transient instructions are never committed to the architectural state, a processor may weaken its security guarantees within the transient domain, allowing instructions to access data they otherwise would not be able to access, *i.e.*, data of another security domain.

A transient-execution attack typically consists of multiple phases. While Canella et al. [50] initially proposed 5 distinct phases and Xiong and Szefer [329] used 3 phases, Canella et al. [47] and Gruss [84] recently used 6 phases to describe the basic idea of a transient-execution attack. However, we only consider 3 phases for simplicity:

1. **Trigger.** The adversary steers the processor to execute certain instructions transiently using a trigger instruction. This trigger instruction can be a mispredicted branch instruction or an instruction triggering a fault, assist, or interrupt.

2. **Access and Send.** In the transient domain, these instructions access secret data and form the sending part of the covert channel.

3. **Receive.** In the architectural domain, the adversary recovers the secret data with the receiving end of the covert channel.

Like the first transient-execution attacks, Meltdown [177] and Spectre [156], we distinguish the different types of transient-execution attacks based on the trigger instruction: While Spectre-type attacks trigger transient execution in the form of speculative execution following a data- or control-flow misprediction, Meltdown-type attacks exploit illegal data flow follow-

ing faults, assists, or other events causing a (partial) flush of the pipeline. With Load Value Injection (LVI), Meltdown-type data leakage effects are reversely exploited to poison transient execution in a victim domain forming another subclass of transient-execution attacks. After the disclosure of the original Meltdown [177] and Spectre [156] paper, different classification schemes have been proposed [50, 127, 251, 329] to shed some light on the jungle of different transient-execution attacks.

Transient-execution attacks can be leveraged within or between different protection domains, e.g., the operating system, virtual machines, or trusted-execution environments such as Intel SGX. However, these domains are not limited to an individual CPU but can span all kinds of devices that are connected to each other [127] as transient-execution attacks have also been demonstrated remotely [264].

In the remainder of this section, we discuss Spectre-, Meltdown- and LVI-type attacks in more detail and the mitigations they require.

3.2.1. Spectre-type Attacks

Spectre-type attacks exploit the transient execution of instructions that follow a control- or data-flow misprediction. The idea is to mistrain the branch-prediction unit such that the processor speculatively executes instructions that do not occur architecturally in the instruction stream. In modern processors, multiple different predictors jointly decide the outcome and the target of a branch (see Section 2.1.2.3). Thus, by poisoning one or multiple prediction units, Spectre-type attacks steer the transient execution to so-called *gadgets*, *i.e.*, code snippets that enable the adversary to expose sensitive data through microarchitectural state changes.

Following the classification scheme by Canella et al. [50], Spectre-type attacks can be classified by the prediction unit they mistrain.

- **Pattern History Table (PHT).** With Spectre-PHT, initially described as Variant 1 [156], the Pattern History Table mispredicts whether a branch should be taken or not taken. The PHT is typically mistrained to take a branch by repeatedly calling the targeted code with arguments that will take a branch, *i.e.*, in-bound values, for a bounds check. Then, the code is executed with an out-of-bounds argument that would architecturally not take the branch. However, the CPU will first mispredict the branch direction and execute the

instructions transiently with the provided out-of-bound values before it detects and corrects the misprediction.

- **Branch Target Buffer (BTB).** With Spectre-BTB, initially described as Variant 2 [156], the Branch Target Buffer is exploited to mispredict the address the control flow should be redirected to. This allows an adversary to basically steer the transient execution to any address in the victim domain, enabling return-oriented programming attack techniques in the transient domain.
- **Return Stack Buffer (RSB).** With the RSB, the return address of a function can be predicted on a `ret` instruction. The RSB stores the location of the most recent `call` instructions. While we suspected that the RSB can be exploited in the original Spectre paper et al. [156], Maisuradze and Rossow [183] and Koruyeh et al. [157] were the first to demonstrate this variant by over- and underflowing the RSB. By poisoning the RSB with incorrect return addresses, the adversary can steer the victim's execution to a gadget leaking sensitive information.
- **Store-to-load Forwarding (STL).** Originally described as Variant 4 [273], Horn exploited the memory disambiguator to mispredict that a load does not depend on an earlier store operation. The load operation is speculatively executed, returning a stale value from the cache.

In addition, there are other hardware predictors whose behavior can influence the transient domain. With *Value Prediction* [170, 171], Lipasti and Shen explore data prediction that tries to predict the actual results of instructions based on previous results. However, while thoroughly researched [145, 216–218, 247] and with value-prediction championships [27], so far, value prediction has not been documented to be implemented in modern microarchitectures. With Zen 3, AMD implements a predictive store forwarding mechanism [11] that predicts dependencies between loads and stores. Before the actual address has been determined, the processor predicts whether store-to-load forwarding will occur between a load and a store based on previous executions.

With Spectre-type attacks, the code gadget performing the secret access and the sending part of the covert channel are always executed in the victim's context. For instance, in a scenario where a user-space application targets the operating system, the access to the secret kernel address is performed speculatively in the kernel's context and, thus, represents a

legitimate access in that domain. However, there are different mistraining strategies that allow the attacker to poison the victim branch [50] executed within the kernel. As branch predictors are typically indexed by the virtual address, it is possible to mistrain the predictor from the same address space or from a different address space controlled by the attacker. In addition, usually, only a subset of the bits of the address is used for the index, allowing to use *shadow branches* whose addresses are congruent to the victim branch. In sandboxing scenarios, e.g., JavaScript in a browser, the adversary targets secrets within the same domain.

While we discuss mitigation strategies against Spectre-type attacks later, there have been various different approaches to automatically find different Spectre gadgets in existing codebases and binaries. Wang et al. [311] used taint tracking, and Bloem et al. [39] used taint analysis and model checking to identify gadgets. Guarnieri et al. [95] used symbolic execution, and Oleksenko et al. [209] used fuzzing techniques to find gadgets.

Kiriansky and Waldspurger [155] used speculative stores for speculative buffer overflows. With speculative probing [76], Göktaş et al. combine Spectre with a memory corruption vulnerability. The attacker first overwrites the victim code pointer before Spectre is used for fault suppression to prevent the application from crashing if the overwritten location is invalid. With SpecROP, Bhattacharyya et al. [37] demonstrated Spectre-style code-reuse attacks. Most Spectre-type attacks have been demonstrated with a cache side-channel to leak from the transient domain [29, 76, 155–157, 183, 273]. Spectre-type attacks have been demonstrated with other side channels as well enabling different gadgets that can be exploited: Bhattacharya et al. [34] and Fustos and Yun [72] use contention-based side channels, we used AMD's way predictor [175], Rensee et al. [240] the μOP cache, and Schwarz et al. [256] used TLB effects on the store buffer. Furthermore, Schwarz et al. [264] used timing differences introduced by the AVX unit. Weber et al. [316] exploits that the AVX unit is resetted faster if an `x87-FPU` instruction is executed. Alternative covert channels enable other Spectre gadgets that have different requirements to be exploited. For instance, our covert channel based on the way predictor lifts the requirement of cache-based covert channels that require shared memory between the adversary and victim domain. Wampler et al. [307] leverages speculative execution to hide malware from static and dynamic analysis.

3.2.1.1. Mitigations

Spectre-type attacks exploit speculative execution and, thus, a fundamental microarchitectural design that increases performance. Mispredictions represent a corner case in speculative execution that should happen only very rarely. While they are explicitly triggered in Spectre-type attacks, they are an expected side effect. Hence, the underlying issue is much more fundamental and cannot be fully mitigated in hardware. Thus, it remains an open issue in the future [191].

To prevent the exploitation of Spectre-type attacks, mitigation strategies on various levels have been suggested. As Canella et al. [48], Xiong et al. [329] and Gruss [84] give a more in-depth overview of these mitigations, we only focus on an excerpt of these.

To prevent the processor from speculating, serializing, or fencing instructions can be inserted to ensure that the following instructions are only executed when all previous instructions are completed. With `lfence` on x86 and `DSB SY` barrier instructions can be used to prevent speculation. These barriers have to be inserted in every potential Spectre gadget to prevent its exploitation. Therefore, automatically detecting these gadgets is an ongoing research effort [39, 95, 209, 311].

Future branch predictor designs could partition [305, 348] or encrypt [165] prediction history on a per-context level. However, these approaches cannot protect against attack variants that use in-place mistraining within the same domain. For existing CPUs, to prevent cross-domain mistraining, Indirect Branch Restricted Speculation (IBRS) does not allow unprivileged code to mistrain privileged code. With Single Thread Indirect Branch Prediction (STIBP), one hardware thread cannot influence the history of the other. The Indirect Branch Predictor Barrier (IBPB) is used during context switches to flush the BTB. With *retpolines* [288], indirect branches are rewritten to special code sequences that use returns. When speculating, the RSB will predict an endless loop until the actual target is known.

3.2.2. Meltdown-type Attacks

Meltdown-type attacks exploit illegal data flow during transient instructions following a fault, assist, or other events, causing a (partial) flush of the pipeline or the selective replay of instructions. While Spectre-type attacks exploit the intended speculative execution of instructions,

Meltdown-type attacks exploit that security guarantees of the architecture are neglected during transient execution. While these microarchitectural design decisions are sensible from a performance point of view and simplicity, they have immense security implications if the microarchitectural state can be inspected. Meltdown-type attacks directly read sensitive data processed in other protection domains without relying on instruction sequences in the victim domain.

In our original Meltdown attack [177], we exploited the fact that data from kernel addresses is forwarded transiently to the unprivileged attacker if the data is stored in the L1 data cache [323]. The unauthorized loads are forwarded to subsequent transient instructions allowing the attacker to establish a cache-based covert channel to exfiltrate the data. When the exception raised by the user-mode access to a kernel page (defined by the *U/S*-bit in the x86 page table entry [118]) is handled on instruction commitment or suppressed by using transactional memory or a misprediction, the microarchitectural state change has already been performed to transmit the secret. With Meltdown, the memory isolation protection of the processor has been *melted down*.

In contrast to Spectre-type attacks where the access to the secret is performed by the victim in the victim's context, Meltdown-type attacks directly bypass the victim's security domain and enable adversaries to leak the data directly. The code sequences accessing the secret are directly controlled by the adversary to leak from other domains, *i.e.*, other user-space applications [177, 251, 262, 276], the operating system or hypervisor [46, 177, 251, 262], virtual machines [262, 276] and trusted-execution environments [237, 262, 293], by picking up data from different buffers and buses.

While the L1 cache has been primarily used to leak data in Meltdown [177] and Foreshadow [293], the complex interaction between different microarchitectural attacks enabled further exploitation of illegal data flows within the transient domain. In the original Meltdown paper [177], we already attributed some leakage to the line-fill buffers (LFB). In subsequent work, the line-fill buffers (LFB) and load ports (LP) have been studied more thoroughly in ZombieLoad [262], RIDL [251], and Medusa [197]. Especially with one variant of ZombieLoad, also known as TSX Asynchronous Abort (TAA), we exploited Intel's transactional memory extension to leak data on CPUs that have been deemed secure against the other described variants. In addition, we demonstrated that the initially proposed mitigations by Intel have been insufficient and could still be exploited. With

CacheOut [252], Van Schaik further explored this variant. These variants have further been described in two addenda by the RIDL authors [249, 250]. With Fallout [46], we showed that we can leak previous stores from the store buffer. With Medusa [197], we developed a fuzzing approach to generate and evaluate new variants that allow leaking from various buffers. Further attacks target floating-point registers [276] and privileged registers [125] on the same CPU core. CrossTalk [237] targets staging buffers that are shared between CPU cores.

We systematize Meltdown-type attacks by classifying them based on their fault condition and back then uncovered two yet undescribed variants that allowed us to bypass protection keys and checks from the `bound` instruction [50]. However, while the proposed canonical naming scheme brought a better structure into the original names of these attacks. This scheme takes an architectural or software viewpoint. Later, Canella et al. [47] looked for similarities in Meltdown-type attacks based on their microarchitectural behavior. With Medusa [197], we use hardware performance counters to find similarities and differences between them to get a better understanding from where values are leaked and which hardware paths they exploit. We show that both, the newly generated and already described variants, fit in the previous naming scheme.

3.2.2.1. Mitigations

In contrast to Spectre-type attacks, Meltdown-type attacks can be mitigated relatively easily on the silicon level. With the assumption that the microarchitectural state cannot be inspected and the security guarantees on the architectural level are met, different processor designs contain different optimizations for speed, simplicity, and energy efficiency. If, thereby, the security guarantees are neglected on the microarchitectural level, Meltdown-type attacks are possible. While Intel seems to be affected by almost all published attacks, some variants are also applicable to microarchitectures from other CPU vendors. While the ARM Cortex-A75 design has been vulnerable to Meltdown [22], we additionally demonstrated that the Samsung Exynos M1 used in the Samsung Galaxy S7 phone has been vulnerable as well [177]. Furthermore, Apple [21] and IBM [226] stated that some of their CPUs are affected by Meltdown as well.

The issue with Meltdown is that despite the failed permission check, the unauthorized load still returns the architectural data value to subsequent

operations that are executed subsequently. The load operation queries the L1 cache and TLB in parallel and continues the load with the corresponding page-frame number (PFN), allowing the load to pick up the correct value from the L1 cache. However, upon detecting the privilege violation, the load could be treated differently or squashed entirely as there is no need for the results to be broadcast to dependent operations. Alternatively, the TLB could return an invalid PFN, e.g., all bits set, or the result of the load could be masked with all bits zeroed out. Furthermore, as the TLB result is required for the tag check, the permission bit is already known, and the load could be handled as if the page is not mapped at all. Canella et al. [49] observed that AMD CPUs stall when a kernel address is accessed without privileges. Moreover, they noted that Intel CPUs containing Meltdown hardware mitigation seem to still execute the load, however, the result is zeroed out. While this prevents the data leakage caused by Meltdown as the adversary will only read a zero value, this behavior still allows certain LVI-type attacks, as we discuss in Section 3.2.3. Henry Wong [323] further analyzed the behavior of Meltdown on a variety of different microarchitectures. On the Intel Core 2, Wong observed the Foreshadow attack where data is forwarded for not-present pages before the Foreshadow paper [293] was public.

These above described design changes to mitigate Meltdown can only be incorporated in new microarchitectures, and, hence, a software workaround is required. For Meltdown to succeed, the kernel address has to be mapped in the virtual address space used by the user-space application. If the kernel page is not mapped while running in user mode, the TLB cannot translate it and, thus, the kernel data cannot be forwarded transiently. In 2017, we proposed the KAISER patch [87] to the Linux kernel to prevent various side-channel attacks that allowed defeating KASLR. By enforcing stronger kernel isolation, kernel pages are not mapped while running in user mode. As this design change of the operating system also defeats Meltdown, it has been adopted by the major operating systems to mitigate Meltdown in software on affected CPUs.

To mitigate other Meltdown-type attacks, Intel introduced or modified instructions that allow flushing different microarchitectural buffers on context switches. For instance, the L1 data cache is flushed when entering or exiting an SGX enclave. Similarly, the hypervisor flushes the L1 data cache when switching between virtual machines. For ZombieLoad and RIDL, the `verw` instruction clears the line-fill buffers and store buffer and is used on every context switch. While more recent CPU generations al-

ready integrate hardware mitigations against ZombieLoad and RIDL, the TAA variant could still leak sensitive information on the Cascade Lake microarchitecture. There, a microcode update introduced an option that effectively disables TSX. In addition, all these mitigations do not prevent attack scenarios where the victim and adversary run on SMT threads of the same core, and, thus, it is recommended to disable hyper-threading.

3.2.3. LVI-type Attacks

LVI-type attacks reversely exploit Meltdown-type leakage effects to inject poisoned values into the victim domain to obtain sensitive information. Instead of a poisoned prediction history that is used in Spectre-type attacks, the transiently injected data values instead trick the victim into exposing its data. Likewise to Spectre-type attacks, the access and sending component of the covert channel is performed by the victim, and, hence, the victim is architecturally allowed to perform that access. In contrast to Meltdown-type attacks, LVI-type attacks require specific code gadgets in the victim domain. However, in contrast to Spectre gadgets, LVI gadgets are much simpler, and depending on the adversary's capabilities, a single load operation can serve as an exploitable LVI gadget.

To exploit an LVI-type attack, the victim has to perform an operation that induces a fault or assist. While these faults can occur during the regular execution of a victim process, the likelihood that the same instruction faults on a regular basis is very low. In our user-to-kernel proof-of-concept, we assume that the targeted kernel page is swappable and artificially clear the accessed bit of the targeted page. As Windows periodically clears the accessed bit of pages, we use the same principle in a user-to-user setting. However, LVI-type attacks can be best performed in scenarios where a malicious operating system targets an Intel SGX enclave to use precise execution control of the enclave [296]. By interrupting the enclave before executing the targeted instruction, the adversary can directly clear the accessed or supervisor bit for the enclave page to trigger the fault when resuming the enclave. Furthermore, the adversary can force the enclave to repeatedly execute the same instruction [296] increasing the chance to inject the malicious data and, thus, to leak the sensitive information. With LVI-NULL, we describe a special case where it is sufficient to inject a zero value into the victim. In Meltdown-resistant CPUs, the results to dependent instructions are zeroed out to prevent data leakage [120, 256] and, therefore, in an LVI-type attack, a zero value would

be transiently injected. While an injected zero value seems rather harmless, when targeting an SGX enclave, a malicious operating system can map the zero page to exploit a transient null-pointer dereference in the enclave.

3.2.3.1. Mitigations

As LVI-type attacks invertedly exploit the leakage effects of Meltdown-type attacks, hardware mitigations of Meltdown-type attacks can be sufficient to eliminate the LVI counterpart. However, in Section 3.2.2.1, we discuss that in case of a faulting load returning a fixed value, e.g., all bits set to 0 or 1, instead of the actual inaccessible value mitigates the data leakage. In an LVI attack, this means that a value of all 0s or 1s is injected into the victim domain. We demonstrated that this behavior on its own can be exploited again by leaking AES-NI keys or mapping the null page in an Intel SGX setting to transiently hijack control flow [294]. Hence, in-silicon mitigations should prevent illegal data flows from loads to dependent operations.

Since hardware mitigations require new microarchitecture revisions, obviously, software-based solutions are necessary to protect existing microarchitectures. The `lfence` instruction acts as a barrier to stop speculative execution. Hence, we suggested to automatically insert an `lfence` instruction as a speculation barrier after every implicit and explicit load operation [294]. However, our proof-of-concept compiler pass shows a non-negligible performance impact between a factor of 2 and 19 [294]. Building upon this approach, Intel developed an optimized LVI mitigation pass for compilers to minimize the number of `lfences` that need to be inserted [116]. The idea is only to insert a barrier between load instructions and subsequent instructions that may transmit the loaded value using a covert channel. Following a similar approach used by Bender et al. [30] to insert fences to enforce certain memory ordering constraints, Intel analyzed the control-flow graph to detect all possible load (source) and transmit instructions (sink). By solving the *minimum multi-cut problem*, the number of fences can be minimized such that every load followed by a transmitter instruction is protected by at least one barrier. While with this algorithm, the minimal number of `lfences` can be inserted, the performance overhead is still significant [164].

3.3. Software-based Microarchitectural Fault Attacks

While traditional fault attacks (see Section 2.3.2) require physical access to the device to induce faults from the outside, *software-based microarchitectural fault attacks* aim to induce faults in microarchitectures solely from software. In this section, we discuss the state-of-the-art of fault attacks inducing bit flips in the main memory and CPU cores.

3.3.1. Rowhammer

In 2014, Kim et al. [153] described that bit flips can be induced in DRAM cells from software by accessing neighboring rows in a high frequency. By forcing repeated memory accesses from DRAM, *i.e.*, using *Flush+Reload*, neighboring rows are frequently opened and closed between refresh cycles, flipping bits by triggering disturbance errors [153]. Seaborn and Dullien [266] made the security implications of the Rowhammer bug clear by gaining kernel privileges from an unprivileged user space program. Subsequent research showed different techniques to exploit the Rowhammer bug in numerous ways [12, 35, 41, 51, 55–57, 69, 70, 88, 91, 134, 139, 152, 161, 163, 169, 174, 179, 220, 224, 228, 238, 241, 283, 300, 318, 327, 333, 339, 346, 347]:

Remote Attacks. Gruss et al. [91] showed that bit flips can be induced within sandboxed JavaScript, enabling remote Rowhammer attacks in the browser. Bosman et al. [41] exploited memory deduplication in combination with Rowhammer to disclose pointers allowing to gain arbitrary read and write access in the Microsoft Edge web browser. Tatar et al. [283] demonstrated remote Rowhammer attacks using RDMA network cards. De Ridder et al. [241] demonstrated Rowhammer attacks are still possible from JavaScript by using a many-sided attack approach.

In a cloud setting, Xiao et al. [327] demonstrated cross-VM Rowhammer attacks. Razavi et al. [238] exploited bit flips in co-located virtual machines to break public-key authentication mechanisms.

With Nethammer (see Chapter 7), we demonstrated remote Rowhammer attacks on common server hardware. Similar to Tatar et al. [283], we invalidate the assumption that Rowhammer requires a local attacker. We

discuss scenarios where an adversary can use uncontrolled bit flips to alter DNS entries or target OCSP servers.

Improving Rowhammer and Rowhammer Techniques. By reverse-engineering the DRAM mapping functions, Pessl et al. [220] allowed to develop faster and more reliable Rowhammer attacks. Aga et al. [12] exploited Intel CAT to accelerate eviction-based Rowhammer attacks. Islam et al. [134] exploited store-to-load forwarding to learn about the physical page mappings, improving Rowhammer attacks. Quao et al. [228] used non-temporal instructions to induce bit flips. Zhang et al. [345, 347] described Rowhammer attacks through implicit memory accesses by the page table walk.

Kwong et al. [161] studied the relationships of data and the induced bit-flip patterns, not to corrupt data in neighboring rows but to actually infer their contents.

We present *one-location hammering* [88] where in contrast to previous attack techniques, we only keep a single row open. In Nethammer (see Chapter 7), we study the effect of memory-replacement policies that enable one-location hammering. Furthermore, we discuss *opcode flipping* as a new exploitation technique by flipping bits in a targeted way in user space binaries [88]. With *memory waylaying*, we exploit system-level optimizations in combination with a side channel to place target pages at attacker-chosen physical locations.

Rowhammer Protections. Cojocar et al. [57] demonstrated Rowhammer attacks on ECC-protected DDR chips. Kim et al. [152] analyzed modern DRAM devices and mitigation techniques and concluded that newer chips are more susceptible to bit flips. Frigo et al. [70] studied TRR in more depth and demonstrated bit flips despite the mitigation on a variety of DIMMs. Walker [306] investigates the physical root causes of bit flips as a starting to point to find effective mitigations for the future.

In Nethammer (see Chapter 7), we already showed that TRR is insufficient to mitigate Rowhammer.

Attacking Cryptographic Implementations. By combining Rowhammer with a *Prime+Probe* cache attack, Bhattacharya et al. [35] recovered cryptographic keys. Poddebniak et al. [224] showed Rowhammer attacks

against EdDSA, and Zeitouni et al. [339] targeted Rowhammer-based physically unclonable functions (PUF). Mus et al. [204] used Rowhammer to recover secret keys from a constant-time implementation of the post-quantum signature scheme LUOV.

Other. Chakraborty et al. [51] attacked the Page Frame Cache (PFC) with Rowhammer. Yao et al. [333] targeted deep neural networks with Rowhammer. Jang et al. [139] used bit flips inside SGX enclaves to mount a denial-of-service attack. Weissman et al. [318] induced bit flips using FPGAs in heterogeneous FPGA-CPU platforms. Van der Veen et al. [300] and Lipp [172] induce bit flips on ARM-based devices. Frigo et al. [69] induced bit flips using GPU primitives on mobile phones.

3.3.2. Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) is an energy-saving technique that reduces energy consumption and temperature. By adjusting voltage and frequency regulators, the core voltage and core frequency can be changed accordingly to the runtime demands. Registers in CPUs are built using flip-flops that only change their states when a clock edge occurs, *i.e.*, the control signal going from '0' to '1' or from '1' to '0'. However, the intermediate combinatorial logic between two registers does not produce their outputs immediately, leaving a small timing delay until the values are propagated. When this timing delay is not taken into account or not met between a clock cycle, the output is not properly latched into the input register, leaving it at a stale value [282]. By either overclocking or undervolting, this timing constraint can be violated, and a fault can be induced.

With CLKSCREW [282], Tang et al. exploited the energy management mechanisms to extract cryptographic keys from TrustZone. Using a malicious kernel driver, they adjusted the frequencies using privileged interfaces to overclock the CPU to induce faults. Qiu et al. [230], on the other hand, manipulated the core voltage to undervolt the ARM processor to recover AES keys processed within TrustZone. Murdock et al. [202] demonstrated with Plundervolt a similar attack on Intel processors. By abusing a not officially documented voltage-scaling interface, they were able to induce predictable faults during SGX enclave computation, to recover keys from AES-NI implementations. Around the same time, Qiu et al. [231, 232], Kenjar et al. [150], and Chen et al. [53] showed similar

attacks targeting Intel SGX. Rabich [235] explored software-based fault attacks on AMD Zen processors. While system crashes were produced, single faults in operations could not be observed.

3.3.3. Circuit Aging

Circuit aging is not only a reliability concern, the security implications of aging need to be taken into account [159] as well. With their MAGIC attack, Karimi et al. [147] simulated specially crafted software sequences over a long period of time to maliciously trigger aging effects in CPU cores that degrade the processor's performance. They observed performance degradation of up to 10.91 % within 4 weeks. Zhao et al. [350] explored the impact of different routing algorithms for aging acceleration in networks-on-chip.

3.4. Software-based Power Side-Channel Attacks

In this section, we want to discuss the state-of-the-art of power side-channel attacks that are conducted using software only. In contrast to traditional power side-channel attacks where physical access or physical proximity to obtain the power measurements is required, these attacks obtain their power information using interfaces exposed to software.

CPU Energy Measurements. To limit, monitor and budget the energy consumption of processors, CPU manufacturers provide interfaces that expose the energy measurements to the operating system. With Intel Running Average Power Limit (RAPL), Intel provides an interface that exposes some of its measurement counters to unprivileged user space [78, 214].

This interface is used to monitor the energy consumption of different workloads [99, 151] or the impact of different software defenses [104]. Only recently, research regarding the security implications of such an interface have been conducted: Fusi [71] used RAPL to attack RSA-16384 but concluded that the sampling rate of RAPL is too low to mount an attack, showing that it is only observable whether branches are taken, and accessed data is cached. Mantel et al. [185] distinguish RSA keys with different Hamming weights using RAPL but do not try to extract keys or perform other concrete attacks. Gao et al. [73] use RAPL in containers to

infer information about the host environment, e.g., co-location of multiple containers.

We advanced the state of the art by analyzing the RAPL interface further. We observed that one can distinguish instructions and also operands and data loaded based on their Hamming weight [176]. We demonstrated attacks against AES-NI, the Linux kernel, and Intel SGX enclaves. We combined the power leakage of Intel RAPL with SGX-step [296] to fingerprint individual instructions executed inside an enclave leaking RSA keys. Furthermore, we established a covert communication channel, even in virtual machines [178] as the RAPL interface has been exposed to Xen guests [324]. Zhang et al. [344] establish a covert communication channel and perform website fingerprinting using Intel RAPL.

Furthermore, other CPU manufacturers, e.g., ARM, NVIDIA [207], IBM POWER [113], Ampere [58], Hygon [319], or Marvell [187], provide different power interfaces as well. However, while we conducted some experiments on AMD CPUs [176], they all have not been studied in detail yet.

Power Analysis on Mobile Devices. Yan et al. [331] monitor system power information on mobile devices to acquire voltage and current, observing a correlation with keystrokes, enabling them to infer password lengths and also distinguish different applications. Qin et al. [229] use the same interfaces to fingerprint websites on mobile devices. Vasilakis [298] characterized the energy consumption of instructions on ARM using power sensors available on the targeted platform.

On-die Power Analysis. O'Flynn and Dewar [208] recorded power measurements using an onboard ADC from the non-secure world to recover secrets processed in the secure world on TrustZone-M. Zhao and Suh [349] use an FPGA to observe a CPU's power consumption on the same SoC to break RSA. Moini et al. [198] targeted BNN accelerators in remote FPGAs. Tian et al. [285] attacked the versatile tensor accelerator in multi-tenant FPGAs. With CAPSULe, Giechaskiel et al. [75] established cross-FPGA covert-channel attacks using leakage of the power supply units. Schellenberg et al. [253] develop an internal sensor in an FPGA to deploy key-recovery attacks.

4
Conclusion

In this thesis, we show that microarchitectural performance optimizations can be exploited from software interfaces on different abstraction levels. From the CPU or DRAM microarchitecture to the microarchitecture on an operating-system level, side channels and fault attacks allow extracting sensitive information over abstraction boundaries. With only software-based approaches, we uncover sensitive information processed by the victim, even in remote scenarios.

With Meltdown and Spectre, the research field of transient-execution attacks emerged that, in contrast to side-channel attacks, allows extracting data directly. To mitigate these attacks, a combination of CPU microcode patches, updates to the operating systems, as well as toolchains and compilers, are necessary; most of them coming with a non-negligible performance impact. However, painting a somewhat darker picture, with a high probability, similar issues exist in any sufficiently enough complex system. Thus, the future will not only bring new attack variants to light but will also require a rethinking of tackling these challenges on the microarchitectural level. In addition, further performance optimizations can and will not only be introduced in the CPU microarchitecture but in every other abstraction layer as well [186, 221–223, 277]. Future research will need to investigate the security implications of these and show if similar or new attacks are possible.

However, the clear definitions of the interfaces that enabled these optimizations in the microarchitectures in the first place allow to either revert the optimizations or safely implement them in new generations without breaking the functionality defined by the architecture. It is astonishing to see that most of these attacks targeting the fine nuances on a microarchitectural level could be mitigated on previous generations with software and firmware updates. However, future designs will need to tackle them on the silicon level.

Bibliography

[1] Andreas Abel and Jan Reineke. “Reverse engineering of cache replacement policies in intel microprocessors and their evaluation”. In: *ISPASS*. 2014.

[2] Andreas Abel and Jan Reineke. “uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures”. In: *ACM ASPLOS*. 2019.

[3] Accardi, Kristen Carlson. *Function Granular KASLR*. 2020. URL: `https://patchwork.kernel.org/project/kernel-hardening/list/?series=354389`.

[4] Onur Acıiçmez. “Yet Another MicroArchitectural Attack: Exploiting I-cache”. In: *CSAW*. 2007.

[5] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. “New Results on Instruction Cache Attacks”. In: *CHES*. 2010.

[6] Onur Acıiçmez and Çetin Kaya Koç. “Trace-Driven Cache Attacks on AES (Short Paper)”. In: *ICICS*. 2006.

[7] Onur Acıiçmez and Cetin Kaya Koç. “Microarchitectural attacks and countermeasures”. In: *Cryptographic Engineering*. 2009.

[8] Onur Acıiçmez and Werner Schindler. “A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL”. In: *CT-RSA*. 2008.

[9] Onur Acıiçmez and Jean-Pierre Seifert. “Cheap Hardware Parallelism Implies Cheap Security”. In: *FDTC*. 2007.

[10] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. “Predicting secret keys via branch prediction”. In: *CT-RSA*. 2007.

[11] Advanced Micro Devices Inc. *Security Analysis of AMD Predictive Store Forwarding*. 2021.

[12] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. “When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks”. In: *HOST*. 2017.

[13] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. “Port Contention for Fun and Profit”. In: *IEEE S&P*. 2018.

[14] Alexa Internet, Inc. *The top 500 sites on the web*. 2016. URL: `http://www.alexa.com/topsites`.

[15] Kamran Ali, Alex X Liu, Wei Wang, and Muhammad Shahzad. “Keystroke recognition using wifi signals”. In: *MobiCom*. 2015.

[16] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. "Amplifying Side Channels Through Performance Degradation". In: *ACSAC*. 2016.

[17] AMD. "Strengthening VM isolation with integrity protection and more". In: *White Paper* (2020).

[18] *AMD Takes Computing to a New Horizon with Ryzen™Processors*. Advanced Micro Devices Inc., 2016. URL: https://www.amd.com/en-us/press-releases/Pages/amd-takes-computing-2016dec13.aspx.

[19] Andrei Frumusanu. *Arm's Cortex-A76 CPU Unveiled: Taking Aim at the Top for 7nm*. 2018. URL: https://www.anandtech.com/show/12785/arm-cortex-a76-cpu-unveiled-7nm-powerhouse/.

[20] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. "On subnormal floating point and abnormal timing". In: *IEEE S&P*. 2015.

[21] Apple Inc. *About speculative execution vulnerabilities in ARM-based and Intel CPUs*. 2018. URL: https://support.apple.com/en-us/HT208394.

[22] ARM. *Cache Speculation Side-channels*. Version 2.4. 2018.

[23] Arm. *Arm Confidential Compute Architecture*. 2021. URL: https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture.

[24] Arm. *Armv8-A Instruction Set Architecture*. 2019.

[25] Arm. *GlobalPlatform based Trusted Execution Environment and TrustZone Ready*. 2013.

[26] Arm Limited. *Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism*. 2018.

[27] Arthur Perais. *Value Prediction Championship*. 2021. URL: https://www.microarch.org/cvp1/.

[28] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. "Software-only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks". In: *IVSW*. 2018.

[29] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. "Speculative Interference Attacks: Breaking Invisible Speculation Schemes". In: *ACM ASPLOS*. 2021.

[30] John Bender, Mohsen Lesani, and Jens Palsberg. "Declarative fence insertion". In: *ACM SIGPLAN*. ACM. 2015.

[31] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit: A small amount of side channel can go a long way". In: *CHES*. 2014.

[32] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[33] Johann Betz, Dirk Westhoff, and Günter Müller. "Survey on covert channels in virtual machines and cloud computing". In: *ETT* (2016).

[34] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. "Branch Prediction Attack on Blinded Scalar Multiplication". In: *IEEE TC* (2019).

[35] Sarani Bhattacharya and Debdeep Mukhopadhyay. "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis". In: *CHES*. 2016.

[36] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. "Hardware prefetchers leak : A revisit of SVF for cache-timing attacks". In: *MICRO*. 2012.

[37] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. "SpecROP: Speculative Exploitation of {ROP} Chains". In: *RAID*. 2020.

[38] Eli Biham and Adi Shamir. "Differential Fault Analysis of Secret Key Cryptosystems". In: *CRYPTO*. 1997.

[39] Roderick Bloem, Swen Jacobs, and Yakir Vizel. "Efficient Information-Flow Verification Under Speculative Execution". In: *Symposium on Automated Technology for Verification and Analysis*. 2019.

[40] Joseph Bonneau and Ilya Mironov. "Cache-collision timing attacks against AES". In: *CHES*. 2006.

[41] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector". In: *IEEE S&P*. 2016.

[42] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *WOOT*. 2017.

[43] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks". In: *USENIX Security Symposium*. 2020.

[44] Billy Brumley and Risto Hakala. “Cache-Timing Template Attacks”. In: *AsiaCrypt*. 2009.

[45] Yuriy Bulygin. “Cpu side-channels vs. virtualization malware: The good, the bad, or the ugly”. In: *ToorCon* (2008).

[46] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. “Fallout: Leaking Data on Meltdown-resistant CPUs”. In: *ACM CCS*. 2019.

[47] Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. “The Evolution of Transient-Execution Attacks”. In: *GLSVLSI*. 2020.

[48] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. “Evolution of Defenses against Transient-Execution Attacks”. In: *GLSVLSI*. 2020.

[49] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. “KASLR: Break It, Fix It, Repeat”. In: *AsiaCCS*. 2020.

[50] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *USENIX Security Symposium*. Extended classification tree and PoCs at https://transient.fail/. 2019.

[51] Anirban Chakraborty, Sarani Bhattacharya, and Debdeep Mukhopadhyay. “ExplFrame: Exploiting Page Frame Cache for Fault Analysis of Block Ciphers”. In: *arXiv:1905.12974* (2019).

[52] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. “SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution”. In: *EuroS&P*. 2019.

[53] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. “VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface”. In: *USENIX Security Symposium*. 2020.

[54] Chih-Cheng Cheng. “The schemes and performances of dynamic branch predictors”. In: *Berkeley Wireless Research Center, Tech. Rep* (2000).

[55] Yueqiang Cheng, Zhi Zhang, and Surya Nepal. “Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation”. In: *arXiv:1802.07060* (2018).

[56] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers". In: *IEEE S&P*. 2020.

[57] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks". In: *IEEE S&P*. 2019.

[58] Ampere Computing. *Ampere AltraTM Linux Kernel Porting Guide*. 2020. URL: `https://github.com/AmpereComputing/ampere-centos-kernel/wiki/Ampere-AltraTM-Linux-Kernel-Porting-Guide`.

[59] Victor Costan and Srinivas Devadas. "Intel SGX Explained". In: *Cryptology ePrint Archive, Report 2016/086* (2016).

[60] Yujie Cui and Xu Cheng. "Abusing Cache Line Dirty States to Leak Information in Commercial Processors". In: (2021). URL: `https://arxiv.org/abs/2104.08559`.

[61] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. "Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks". In: *CHES*. 2018.

[62] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. "Secure TLBs". In: *Proceedings of the International Symposium on Computer Architecture*. ISCA. 2019.

[63] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX". In: *USENIX Security Symposium*. 2017.

[64] Jake Edge. *Kernel address space layout randomization*. 2013. URL: `https://lwn.net/Articles/569635/`.

[65] Dmitry Evtyushkin and Dmitry Ponomarev. "Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations". In: *ACM CCS*. 2016.

[66] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Covert channels through branch predictors: a feasibility study". In: *HASP*. 2015.

[67] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR". In: *MICRO*. 2016.

[68] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". In: *ACM ASPLOS*. 2018.

[69] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU". In: *S&P*. 2018.

[70] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "TRRespass: Exploiting the Many Sides of Target Row Refresh". In: *S&P*. 2020.

[71] Matteo Fusi. *Information-Leakage Analysis Based on Hardware Performance Counters*. 2017.

[72] Jacob Fustos and Heechul Yun. "SpectreRewind: Leaking Secrets to Past Instructions". In: *ASHES*. 2020.

[73] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. "ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds". In: *DSN*. 2017.

[74] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware". In: *Journal of Cryptographic Engineering* (2016).

[75] Ilias Giechaskiel, Kasper Rasmussen, and Jakub Szefer. "CAPSULe: Cross-FPGA Covert-Channel Attacks through Power Supply Unit Leakage". In: *IEEE S&P*. 2020.

[76] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. "Speculative Probing: Hacking Blind in the Spectre Era". In: *ACM CCS*. 2020.

[77] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache Attacks on Intel SGX". In: *EuroSec*. 2017.

[78] Corey Gough, Ian Steiner, and Winston Saunders. *Energy Efficient Servers*. Apress, 2015.

[79] Ben Gras and Kaveh Razavi. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *NDSS*. 2017.

[80] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *USENIX Security Symposium*. 2018.

[81] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think". In: *USENIX Security Symposium*. 2017.

[82] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. “Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme”. In: *CHES*. 2016.

[83] Daniel Gruss. “Software-based Microarchitectural Attacks”. PhD thesis. Graz University of Technology, 2017.

[84] Daniel Gruss. “Transient-Execution Attacks”. Habilitation. Graz University of Technology, 2020.

[85] Daniel Gruss, David Bidner, and Stefan Mangard. “Practical Memory Deduplication Attacks in Sandboxed JavaScript”. In: *ESORICS*. 2015.

[86] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. “Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory”. In: *USENIX Security Symposium*. 2017.

[87] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. “KASLR is Dead: Long Live KASLR”. In: *ESSoS*. 2017.

[88] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O’Connell, Wolfgang Schoechl, and Yuval Yarom. “Another Flip in the Wall of Rowhammer Defenses”. In: *IEEE S&P*. 2018.

[89] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *ACM CCS*. 2016.

[90] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *ACM CCS*. 2016.

[91] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA*. 2016.

[92] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA*. 2016.

[93] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. “Use-after-freemail: Generalizing the use-after-free problem and applying it to email services”. In: *AsiaCCS*. 2018.

[94] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

[95] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. "SPECTECTOR: Principled Detection of Speculative Information Flows". In: *IEEE S&P*. 2020.

[96] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice". In: *IEEE S&P*. 2011.

[97] Berk Gulmezoglu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cross-VM cache attacks on AES". In: *IEEE TMSCS* (2016).

[98] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "A Faster and More Realistic Flush+Reload Attack on AES". In: *COSADE*. 2015.

[99] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. "Measuring energy consumption for short code paths using RAPL". In: *ACM SIGMETRICS* (2012).

[100] Chris Hall, Ian Goldberg, and Bruce Schneier. "Reaction attacks against several public-key cryptosystem". In: *ICICS*. 1999.

[101] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. "Side channels in cloud services, the case of deduplication in cloud storage". In: *IEEE S&P* (2010).

[102] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. "Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters". In: *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2020.

[103] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufmann, 2017.

[104] Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat, and Timo Hönig. "The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level". In: *EuroSys*. 2021.

[105] Sebastien Hily, Zhongying Zhang, and Per Hammarlund. *Resolving false dependencies of speculative load instructions*. US Patent 7,603,527. 2009.

[106] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. "The microarchitecture of the Pentium 4 processor". In: *Intel Technology Journal*. 2001.

[107] Rodney E Hooker and Colin Eddy. *Store-to-load forwarding based on load/store address computation source information comparisons*. US Patent 8,533,438. 2013.

[108] Branden Hookway. *Interface*. MIT Press, 2014.
[109] Jann Horn. *Speculative Execution, Variant 4: Speculative Store Bypass*. 2018.
[110] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR". In: *IEEE S&P*. 2013.
[111] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX". In: *CHES*. 2020.
[112] IBM. *IBM Secure Execution for Linux*. 2020.
[113] IBM. *POWER9 Processor User's Manual*. 2.0. 2018.
[114] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cache Attacks Enable Bulk Key Recovery on the Cloud". In: *CHES*. 2016.
[115] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud". In: *Cryptology ePrint Archive, Report 2015/898* (2015).
[116] Intel. *Affected Processors: Transient Execution Attacks*. 2020. URL: https://software.intel.com/security-software-guidance/best-practices/optimized-mitigation-approach-load-value-injectioOn.
[117] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2019.
[118] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2019.
[119] Intel. *Intel Analysis of Speculative Execution Side Channels*. 2018. URL: https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf.
[120] Intel. *Intel Analysis of Speculative Execution Side Channels*. Revision 4.0. 2018.
[121] Intel. *Intel Software Guard Extensions (Intel SGX)*. 2019. URL: https://software.intel.com/en-us/sgx.
[122] Intel. *Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort / CVE-2019-11135 / INTEL-SA-00270*. 2019. URL: https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/intel-tsx-asynchronous-abort.html.

[123] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*. 2012.

[124] Intel. *L1D Eviction Sampling / CVE-2020-0549 / INTEL-SA-00329*. 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/l1d-eviction-sampling.html.

[125] Intel. *Q2 2018 Speculative Execution Side Channel Update*. 2018.

[126] Intel. *Speculative Execution Side Channel Mitigations*. Revision 3.0. 2018.

[127] Intel Corporation. *Refined Speculative Execution Terminology*. 2020. URL: https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology.

[128] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cross processor cache attacks". In: *AsiaCCS*. 2016.

[129] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES". In: *IEEE S&P*. 2015.

[130] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Systematic reverse engineering of cache slice selection in Intel processors". In: *DSD*. 2015.

[131] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Know Thy Neighbor: Crypto Library Detection in Cloud". In: *PETS* (2015).

[132] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Lucky 13 Strikes Back". In: *AsiaCCS*. 2015.

[133] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES". In: *RAID*. 2014.

[134] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks". In: *USENIX Security Symposium*. 2019.

[135] Akanksha Jain and Calvin Lin. "Cache Replacement Policies". In: *Synthesis Lectures on Computer Architecture* (2019).

[136] Himanshi Jain, D Anthony Balaraju, and Chester Rebeiro. "Spy Cartel: Parallelizing Evict+ Time-Based Cache Attacks on Last-Level Caches". In: *Journal of Hardware and Systems Security* (2019).

[137] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. "High performance cache replacement using re-reference interval prediction (RRIP)". In: *ACM SIGARCH* (2010).

[138] Suman Jana and Vitaly Shmatikov. "Memento: Learning secrets from process footprints". In: *IEEE S&P*. 2012.

[139] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. "SGX-Bomb: Locking Down the Processor via Rowhammer Attack". In: *SysTEX*. 2017.

[140] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX". In: *ACM CCS*. 2016.

[141] Jedec Solid State Technology Association. *Low Power Double Data Rate 3*. 2013. URL: `http://www.jedec.org/standards-documents/docs/jesd209-4a`.

[142] Yvon Jegou and Olivier Temam. "Speculative prefetching". In: *ICS*. 1993.

[143] Daniel A Jiménez and Calvin Lin. "Dynamic branch prediction with perceptrons". In: *IEEE HPCA*. 2001.

[144] David R Kaeli and Philip G Emma. "Branch history table prediction of moving target branches due to subroutine returns". In: *ACM SIGARCH* (1991).

[145] Kleovoulos Kalaitzidis and André Seznec. "Leveraging Value Equality Prediction for Value Speculation". In: *ACM TACO* (2020).

[146] David Kaplan, Jeremy Powell, and Tom Woller. "AMD Memory Encryption". In: *White paper* (2016).

[147] Naghmeh Karimi, Arun Karthik Kanuparthi, Xueyang Wang, Ozgur Sinanoglu, and Ramesh Karri. "Magic: Malicious aging in circuits/cores". In: *ACM TACO* (2015).

[148] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. "A high-resolution side-channel attack on last-level cache". In: *DAC*. 2016.

[149] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. "Side channel cryptanalysis of product ciphers". In: *ESORICS*. 1998.

[150] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. "V0LTpwn: Attacking x86 Processor Integrity from Software". In: *USENIX Security Symposium*. 2020.

[151] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ToMPECS* (2018).

[152] Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. “Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques”. In: *ISCA*. 2020.

[153] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors”. In: *ISCA*. 2014.

[154] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. “Exploiting the dram microarchitecture to increase memory-level parallelism”. In: *arXiv:1805.01966* (2018).

[155] Vladimir Kiriansky and Carl Waldspurger. “Speculative Buffer Overflows: Attacks and Defenses”. In: *arXiv:1807.03757* (2018).

[156] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *IEEE S&P*. 2019.

[157] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. “Spectre Returns! Speculation Attacks using the Return Stack Buffer”. In: *WOOT*. 2018.

[158] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs”. In: *EuroS&P*. 2020.

[159] Daniel Kraak, Mottaqiallah Taouil, Said Hamdioui, Pieter Weckx, Francky Catthoor, Abhijit Chatterjee, Adit Singh, Hans-Joachim Wunderlich, and Naghmeh Karimi. “Device aging: A reliability and security concern”. In: *IEEE ETC*. 2018.

[160] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “NetCAT: Practical Cache Attacks from the Network”. In: *S&P*. 2020.

[161] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. “RAMBleed: Reading Bits in Memory Without Accessing Them”. In: *IEEE S&P*. 2020.

[162] Butler W Lampson. “A note on the confinement problem”. In: *Communications of the ACM* (1973).

[163] Mark Lanteigne. *How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware*. 2016. URL: `http://www.thirdio.com/rowhammer.pdf`.

[164] Michael Larabel. *The Brutal Performance Impact From Mitigating The LVI Vulnerability*. 2020.

[165] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. "Securing Branch Predictors with Two-Level Encryption". In: *ACM TACO* (2020).

[166] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing". In: *USENIX Security Symposium*. 2017.

[167] Levi Norman. *Latency: The Heartbeat of a Solid State Disk*. 2010.

[168] David Levinthal. *Performance Analysis Guide for Intel*. 2009.

[169] Chulseung Lim, Kyungbae Park, Geunyong Bak, Donghyuk Yun, Myungsang Park, Sanghyeon Baeg, Shi-Jie Wen, and Richard Wong. "Study of proton radiation effect to row hammer fault in DDR4 SDRAMs". In: *Microelectronics Reliability* (2018).

[170] Mikko H Lipasti and John Paul Shen. "Exceeding the dataflow limit via value prediction". In: *MICRO*. 1996.

[171] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. "Value locality and load value prediction". In: *ACM SIGPLAN* (1996).

[172] Moritz Lipp. *Cache Attacks and Rowhammer on ARM*. Master Thesis, Graz University of Technology. 2016.

[173] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. "Practical Keystroke Timing Attacks in Sandboxed JavaScript". In: *ESORICS*. 2017.

[174] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security Symposium*. 2016.

[175] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors". In: *AsiaCCS*. 2020.

[176] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. "PLATYPUS: Software-based Power Side-Channel Attacks on x86". In: *IEEE S&P*. 2021.

[177] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security Symposium*. 2018.

[178] Moritz Lipp, Michael Schwarz, Andreas Kogler, and Daniel Gruss. *Attacking CPUs with Power Side Channels from Software: Warum leaked hier Strom?* rC3. 2020.

[179] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. "Nethammer: Inducing Rowhammer Faults through Network Requests". In: *SILM Workshop*. 2020.

[180] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *IEEE S&P*. 2015.

[181] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. "A survey of microarchitectural side-channel vulnerabilities, attacks and defenses in cryptography". In: *arXiv:2103.14244* (2021).

[182] Venkateswara Madduri, Jonathan Combs, James E Phillips, Stephen J Robinson, James D Allen, and Jonathan J Tyler. *Microarchitecture for eliminating MOV operations*. US Patent 9,454,371. 2016.

[183] G. Maisuradze and C. Rossow. "ret2spec: Speculative Execution Using Return Stack Buffers". In: *ACM CCS*. 2018.

[184] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. 2008.

[185] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. "How Secure is Green IT? The Case of Software-Based Energy Side Channels". In: *ESORICS*. 2018.

[186] Jan Kasper Martinsen, Hakan Håkan, and Anders Isberg. "Using speculation to enhance javascript performance in web applications". In: *IEEE Internet Computing* (2012).

[187] Marvell. *tx2mon*. 2020. URL: `https://github.com/Marvell-SPBU/tx2mon`.

[188] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "Reverse Engineering Intel Complex Addressing Using Performance Counters". In: *RAID*. 2015.

[189] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "C5: Cross-Cores Cache Covert Channel". In: *DIMVA*. 2015.

[190] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: *NDSS*. 2017.

[191] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. “Spectre is here to stay: An analysis of side-channels and speculative execution”. In: *arXiv:1902.05178* (2019).

[192] Vahid Meraji and Hadi Soleimany. “Evict+ Time Attack on Intel CPUs without Explicit Knowledge of Address Offsets.” In: *ISeCure* (2021).

[193] Sparsh Mittal. “A survey of techniques for architecting TLBs”. In: *Concurrency and computation: practice and experience* (2017).

[194] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MemJam: A False Dependency Attack against Constant-Time Crypto Implementations in SGX”. In: *CT-RSA*. 2018.

[195] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “CacheZoom: How SGX amplifies the power of cache attacks”. In: *CHES*. 2017.

[196] Daniel Moghimi. “Data Sampling on MDS-resistant 10th Generation Intel Core (Ice Lake)”. In: *arXiv:2007.07428* (2020).

[197] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. “Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis”. In: *USENIX Security Symposium*. 2020.

[198] Shayan Moini, Shanquan Tian, Daniel Holcomb, Jakub Szefer, and Russell Tessier. “Power Side-Channel Attacks on BNN Accelerators in Remote FPGAs”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2021).

[199] John Monaco. “SoK: Keylogging Side Channels”. In: *IEEE S&P*. 2018.

[200] Robert Morgan. *Building an optimizing compiler*. Digital Press, 1998.

[201] David Mulnix. “Intel Xeon processor scalable family technical overview”. In: *Intel Corp* (2017).

[202] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. “Plundervolt: Software-based fault injection attacks against Intel SGX”. In: *IEEE S&P*. 2020.

[203] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. In: *IEEE S&P*. 2020.

[204] Koksal Mus, Saad Islam, and Berk Sunar. “QuantumHammer: A Practical Hybrid Attack on the LUOV Signature Scheme”. In: *ACM SIGSAC*. 2020.

[205] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. “Winter is here!

A decade of cache-based side-channel attacks, detection & mitigation for RSA". In: *Information Systems* (2020).

[206] Michael Neve and Jean-Pierre Seifert. "Advances on Access-Driven Cache Attacks on AES". In: *SAC*. 2006.

[207] NVIDIA. *Jetson TX2: Thermal Design Guide*. 2017.

[208] Colin O'Flynn and Alex Dewar. "On-Device Power Analysis Across Hardware Security Domains". In: *CHES* (2019).

[209] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. "Bringing Spectre-type vulnerabilities to the surface". In: *USENIX Security*. 2020.

[210] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications". In: *ACM CCS*. 2015.

[211] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES". In: *CT-RSA*. 2006.

[212] Rodney Owens and Weichao Wang. "Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines". In: *IEEE IPCCC*. 2011.

[213] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. "Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical". In: *arXiv:2103.03443* (2021).

[214] Jacob Pan. *RAPL (Running Average Power Limit) driver*. 2013. URL: `https://lwn.net/Articles/545745/`.

[215] PaX Team. *Address space layout randomization (ASLR)*. 2003.

[216] Arthur Perais and André Seznec. "BeBoP: A cost effective predictor infrastructure for superscalar value prediction". In: *IEEE HPCA*. 2015.

[217] Arthur Perais and André Seznec. "EOLE: Paving the way for an effective implementation of value prediction". In: *ISCA*. 2014.

[218] Arthur Perais and André Seznec. "Practical data value speculation for future high-end processors". In: *IEEE HPCA*. 2014.

[219] Colin Percival. "Cache missing for fun and profit". In: *BSDCan*. 2005.

[220] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *USENIX Security Symposium*. 2016.

[221] Christopher JF Pickett and Clark Verbrugge. "Return value prediction in a Java virtual machine". In: *VPW*. 2004.

[222] Christopher JF Pickett and Clark Verbrugge. “Software thread level speculation for the Java language and virtual machine environment”. In: *LCPC*. 2005.

[223] Filip Pizlo. *Speculation in JavaScriptCore*. 2020. URL: https://webkit.org/blog/10308/speculation-in-javascriptcore/.

[224] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. “Attacking deterministic signature schemes using fault attacks”. In: *EuroS&P*. 2018.

[225] Joop van de Pol, Nigel P Smart, and Yuval Yarom. “Just a little bit more”. In: *CT-RSA*. 2015.

[226] *Potential Impact on Processors in the POWER Family*. IBM, 2018. URL: https://www.ibm.com/blogs/psirt/potential-impact-processors-power-family/.

[227] Michael D Powell, Amit Agarwal, TN Vijaykumar, Babak Falsafi, and Kaushik Roy. “Reducing set-associative cache energy via way-prediction and selective direct-mapping”. In: *MICRO*. 2001.

[228] Rui Qiao and Mark Seaborn. “A New Approach for Rowhammer Attacks”. In: *IEEE HOST*. 2016.

[229] Yi Qin and Chuan Yue. “Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7”. In: *TrustCom/BigDataSE*. 2018.

[230] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. “VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies”. In: *ACM CCS*. 2019.

[231] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. “VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults”. In: *AsianHOST*. 2019.

[232] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, Ruidong Tian, Chunlu Wang, and Gang Qu. “VoltJockey: A New Dynamic Voltage Scaling based Fault Injection Attack on Intel SGX”. In: *IEEE TCADICS* (2020).

[233] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, Ruidong Tian, Chunlu Wang, and Gang Qu. “VoltJockey: A New Dynamic Voltage Scaling based Fault Injection Attack on Intel SGX”. In: *IEEE TCAD* (2020).

[234] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. “Adaptive insertion policies for high performance caching”. In: *ACM SIGARCH* (2007).

[235] Anja Rabich, Thomas Eisenbarth, and Luca Wilke. “Software-based Undervolting Faults in AMD Zen Processors”. Thesis.

[236] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CROSSTALK: Speculative Data Leaks Across Cores Are Real". In: *IEEE S&P*. 2021.

[237] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real". In: *IEEE S&P*. 2021.

[238] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. "Flip Feng Shui: Hammering a Needle in the Software Stack". In: *USENIX Security Symposium*. 2016.

[239] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. "Side channel attack on NoC-based MPSoCs are practical: NoC Prime+Probe attack". In: *SBCCI*. 2016.

[240] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. "I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches". In: (2021).

[241] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. "SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript". In: *USENIX Security Symposium*. 2021.

[242] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds". In: *ACM CCS*. 2009.

[243] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. "Reverse Engineering the Stream Prefetcher for Profit". In: *SILM Workshop*. 2020.

[244] Kevin W Rudd. "Efficient exception handling techniques for high-performance processor architectures". In: *Departments of Electrical Engineering and Computer Science, Stanford University, Technical Report CSL-TR-97-732* (1997).

[245] Jeff Rupley, Brad Burgess, Brian Grayson, and Gerald D Zuraski. "Samsung M3 processor". In: *MICRO* (2019).

[246] Samsung. *Samsung Z-SSD SZ985: Ultra-low Latency SSD for Enterprise and Data Centers*. 2018.

[247] Yiannakis Sazeides and James E Smith. "The predictability of data values". In: *MICRO*. 1997.

[248] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. *SGAxe: How SGX Fails in Practice*. 2020.

[249] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cris-

tiano Giuffrida. *Addendum 2 to RIDL: Rogue In-flight Data Load*. 2020.

[250] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. *Addendum to RIDL: Rogue In-flight Data Load*. 2019.

[251] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-flight Data Load". In: *IEEE S&P*. 2019.

[252] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. "CacheOut: Leaking Data on Intel CPUs via Cache Evictions". In: *IEEE S&P*. 2020.

[253] Falk Schellenberg, Dennis RE Gnad, Amir Moradi, and Mehdi B Tahoori. "An inside job: Remote power analysis attacks on FPGAs". In: *IEEE Design & Test* (2021).

[254] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, 2007.

[255] David Schor. *Skylake (client)*. 2018. URL: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)`.

[256] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs". In: *arXiv:1905.05725* (2019).

[257] Michael Schwarz, Daniel Gruss, Moritz Lipp, Maurice Clémentine, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features". In: *AsiaCCS* (2018).

[258] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *DIMVA*. 2017.

[259] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. "ConTExT: A Generic Approach for Mitigating Spectre". In: *NDSS*. 2020.

[260] Michael Schwarz, Moritz Lipp, and Daniel Gruss. "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks". In: *NDSS*. 2018.

[261] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks". In: *NDSS*. 2018.

[262] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *ACM CCS*. 2019.

[263] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *FC*. 2017.

[264] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network". In: *ESORICS*. 2019.

[265] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Abusing Intel SGX to conceal cache attacks". In: *Cybersecurity* (2020).

[266] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges". In: *Black Hat Briefings*. 2015.

[267] O Seongil, Young Hoon Son, Nam Sung Kim, and Jung Ho Ahn. "Row-buffer decoupling: A case for low-latency DRAM microarchitecture". In: *ISCA*. 2014.

[268] John Paul Shen and Mikko H Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, 2013.

[269] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. "Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage". In: *ACM SIGSAC*. 2018.

[270] Olin Sibert, Phillip Porras, and Robert Lindell. "The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems". In: *IEEE S&P*. 1995.

[271] Wei Song and Peng Liu. "Dynamically Finding Minimal Eviction Sets Can be Quicker Than You Think for Side-Channel Attacks Against the LLC". In: *RAID*. 2019.

[272] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 2011.

[273] *Spectre Variant 4*. 2018. URL: `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`.

[274] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. "ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android". In: *AsiaCCS*. 2018.

[275] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. "Systematic classification of side-channel attacks: a

case study for mobile devices". In: *IEEE Communications Surveys & Tutorials* (2017).

[276] Julian Stecklina and Thomas Prescher. "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels". In: *arXiv:1806.07480* (2018).

[277] Lixin Su and Mikko H Lipasti. "Speculative optimization using hardware-monitored guarded regions for Java virtual machines". In: *IEEE VEE*. 2007.

[278] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. "Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds". In: *NDSS*. 2018.

[279] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. "Memory Deduplication as a Threat to the Guest OS". In: *EuroSys*. 2011.

[280] Jakub Szefer. "Survey of microarchitectural side and covert channels, attacks, and defenses". In: *Journal of Hardware and Systems Security* (2019).

[281] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[282] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *USENIX Security Symposium*. 2017.

[283] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Throwhammer: Rowhammer Attacks over the Network and Defenses". In: *USENIX ATC*. 2018.

[284] Elvira Teran, Zhe Wang, and Daniel A Jiménez. "Perceptron learning for reuse prediction". In: *MICRO*. 2016.

[285] Shanquan Tian, Shayan Moini, Adam Wolnikowski, Daniel Holcomb, Russell Tessier, and Jakub Szefer. "Remote Power Attacks on the Versatile Tensor Accelerator in Multi-Tenant FPGAs". In: *IEEE FCCM*. 2021.

[286] Robert M Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM Journal of Research and Development* (1967).

[287] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures". In: *Journal of Cryptology* (2010).

[288] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018. URL: https://support.google.com/faqs/answer/7625886.

[289] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. "Exploiting hardware performance counters". In: *FDTC*. 2008.

[290] Vladimir Uzelac and Aleksandar Milenkovic. "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures". In: *IEEE ISPASS*. 2009.

[291] V8 team. *v8 - Documentation*. 2019. URL: https://v8.dev/docs.

[292] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow". In: *MICRO* (2019).

[293] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security Symposium*. 2018.

[294] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *IEEE S&P*. 2020.

[295] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *ACM CCS*. 2018.

[296] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *SysTEX*. 2017.

[297] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution". In: *USENIX Security Symposium*. 2017.

[298] Evangelos Vasilakis. "An Instruction Level Energy Characterization of ARM Processors". In: *FORTH-ICS/TR-450* (2015).

[299] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. "Drammer. Deterministic Rowhammer Attacks on Mobile Platforms". In: *ACM CCS*. 2016.

[300] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms". In: *ACM CCS*. 2016.

[301] Bill Venners. "The java virtual machine". In: *Java and the Java Virtual Machine: Definition, Verification, Validation* (1998).

[302] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. "CacheQuery: Learning Replacement Policies from Hardware Caches". In: *PLDI*. 2020.

[303] Pepe Vila, Boris Köpf, and Jose Morales. "Theory and Practice of Finding Eviction Sets". In: *IEEE S&P*. 2019.

[304] Lucian N Vintan and Mihaela Iridon. "Towards a high performance neural branch predictor". In: *IEEE IJCNN*. 1999.

[305] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. "BRB: Mitigating Branch Predictor Side-Channels". In: *IEEE HPCA*. 2019.

[306] Andrew J Walker, Sungkwon Lee, and Dafna Beery. "On DRAM Rowhammer and the Physics of Insecurity". In: *IEEE TED*. 2021.

[307] Jack Wampler, Ian Martiny, and Eric Wustrow. "ExSpectre: Hiding Malware in Speculative Execution." In: *NDSS*. 2019.

[308] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. "Volcano: Stateless Cache Side-channel Attack by Exploiting Mesh Interconnect". In: *arXiv:2103.04533* (2021).

[309] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. "Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries". In: *NDSS*. 2019.

[310] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. "PAPP: Prefetcher-Aware Prime and Probe Side-channel Attack". In: *DAC*. 2019.

[311] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. "oo7: Low-overhead Defense against Spectre attacks via Program Analysis". In: *Transactions on Software Engineering* (2019).

[312] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. "DRAMDig: a knowledge-assisted tool to uncover DRAM address mapping". In: *DAC*. 2020.

[313] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A

Gunter. “Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX”. In: *ACM CCS*. 2017.

[314] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. “Timing channel protection for a shared memory controller”. In: *IEEE HPCA*. 2014.

[315] Zhenghong Wang and Ruby B Lee. “Covert and Side Channels due to Processor Architecture”. In: *ACSAC*. 2006.

[316] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. “Osiris: Automated Discovery Of Microarchitectural Side Channels”. In: *USENIX Security Symposium*. 2021.

[317] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. “Single Trace Attack Against RSA Key Generation in Intel SGX SSL”. In: *AsiaCCS*. 2018.

[318] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. “JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms”. In: *arXiv:1912.11523* (2019).

[319] Pu Wen. *Add support for Hygon Fam 18h (Dhyana) RAPL*. 2019. URL: `https://patchwork.kernel.org/patch/11123607/`.

[320] Felix Wilhelm. *PoC for breaking hypervisor ASLR using branch target buffer collisions*. 2016. URL: `https://github.com/felixwilhelm/mario_baslr`.

[321] Henry Wong. *Intel Ivy Bridge Cache Replacement Policy*. Retrieved on July 16, 2015. URL: `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`.

[322] Henry Wong. *Microbenchmarking Return Address Branch Prediction*. Retrieved on September 17, 2018. URL: `http://blog.stuffedcow.net/2018/04/ras-microbenchmarks/`.

[323] Henry Wong. *The Microarchitecture Behind Meltdown*. 2018. URL: `http://blog.stuffedcow.net/2018/05/meltdown-microarchitecture/`.

[324] xenbits. *Information leak via power sidechannel*. 2020. URL: `https://xenbits.xen.org/xsa/advisory-351.html`.

[325] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. “Security implications of memory deduplication in a virtualized environment”. In: *IEEE DSN*. 2013.

[326] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. “Stacco: Differentially Analyzing Side-channel Traces for Detect-

ing SSL/TLS Vulnerabilities in Secure Enclaves". In: *ACM CCS*. 2017.

[327] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation". In: *USENIX Security Symposium*. 2016.

[328] Wenjie Xiong, Stefan Katzenbeisser, and Jakub Szefer. "Leaking Information Through Cache LRU States in Commercial Processors and Secure Caches". In: *IEEE TC* (2021).

[329] Wenjie Xiong and Jakub Szefer. "Survey of Transient Execution Attacks". In: *arXiv:2005.13435* (2020).

[330] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *IEEE S&P*. 2015.

[331] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. "A Study on Power Side Channels on Mobile Devices". In: *ACM Internetware*. 2015.

[332] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. "Attack directories, not caches: Side channel attacks in a non-inclusive world". In: *IEEE S&P*. 2019.

[333] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips". In: *USENIX Security Symposium*. 2020.

[334] Yuval Yarom and Naomi Benger. "Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack". In: *Cryptology ePrint Archive, Report 2014/140* (2014).

[335] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014.

[336] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. "Mapping the Intel Last-Level Cache". In: *Cryptology ePrint Archive, Report 2015/905* (2015).

[337] Tse-Yu Yeh and Yale N Patt. "Two-level adaptive training branch prediction". In: *MICRO*. 1991.

[338] Andreas Zankl, Hermann Seuschek, Gorka Irazoqui, and Berk Gulmezoglu. "Side-channel Attacks in the Internet of Things: Threats and Challenges". In: *Research Anthology on Artificial Intelligence Applications in Security*. 2021.

[339] Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. “It’s Mammer Time: How to Attack (Rowhammer-based) DRAM-PUFs”. In: *DAC*. 2018.

[340] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Mingsheng Wang, and Peng Liu. “A Comprehensive Study of Co-residence Threat in Multi-tenant Public PaaS Clouds”. In: *ICICS*. 2016.

[341] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. “Return-oriented Flush-Reload Side Channels on ARM and their Implications for Android Devices”. In: *ACM CCS*. 2016.

[342] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Cross-Tenant Side-Channel Attacks in PaaS Clouds”. In: *ACM CCS*. 2014.

[343] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Cross-VM Side Channels and Their Use to Extract Private Keys”. In: *ACM CCS*. 2012.

[344] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. “Red Alert for Power Leakage: Exploiting Intel RAPL-Induced Side Channels”. In: *AsiaCCS*. 2021.

[345] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, and Zhi Wang. *TeleHammer: A Formal Model of Implicit Rowhammer*. 2020. eprint: 1912.03076.

[346] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, and Zhi Wang. “TeleHammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses”. In: *arXiv:1912.03076* (2019).

[347] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. “PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses”. In: *MICRO*. 2020.

[348] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. “A lightweight isolation mechanism for secure branch predictors”. In: *arXiv:2005.08183* (2020).

[349] Mark Zhao and G Edward Suh. “FPGA-based Remote Power Side-Channel Attacks”. In: *IEEE S&P*. 2018.

[350] Yinyuan Zhao, Xiaohang Wang, Yingtao Jiang, Liang Wang, Amit Kumar Singh, Letian Huang, and Mei Yang. “An enhanced planned obsolescence attack by aging networks-on-chip”. In: *JSA* (2021).

Part II.

Publications

5

Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors

Publication Data

Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors". In: *AsiaCCS*. 2020

Contributions

Main author.

Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors

Moritz Lipp[1], Vedad Hadžić[1], Michael Schwarz[1], Arthur Perais[2], Clémentine Maurice[3], Daniel Gruss[1]

[1] Graz University of Technology [2] Unaffiliated [3] Univ Rennes, CNRS, IRISA

Abstract

To optimize the energy consumption and performance of their CPUs, AMD introduced a way predictor for the L1-data (L1D) cache to predict in which cache way a certain address is located. Consequently, only this way is accessed, significantly reducing the power consumption of the processor.

In this paper, we are the first to exploit the cache way predictor. We reverse-engineered AMD's L1D cache way predictor in microarchitectures from 2011 to 2019, resulting in two new attack techniques. With Collide+Probe, an attacker can monitor a victim's memory accesses without knowledge of physical addresses or shared memory when time-sharing a logical core. With Load+Reload, we exploit the way predictor to obtain highly-accurate memory-access traces of victims on the same physical core. While Load+Reload relies on shared memory, it does not invalidate the cache line, allowing stealthier attacks that do not induce any last-level-cache evictions.

We evaluate our new side channel in different attack scenarios. We demonstrate a covert channel with up to 588.9 kB/s, which we also use in a Spectre attack to exfiltrate secret data from the kernel. Furthermore, we present a key-recovery attack from a vulnerable cryptographic implementation. We also show an entropy-reducing attack on ASLR of the kernel of a fully patched Linux system, the hypervisor, and our own address space from JavaScript. Finally, we propose countermeasures in software and hardware mitigating the presented attacks.

1. Introduction

With caches, out-of-order execution, speculative execution, or simultaneous multithreading (SMT), modern processors are equipped with numerous features optimizing the system's throughput and power consumption. Despite their performance benefits, these optimizations are often not designed with a central focus on security properties. Hence, microarchitectural attacks have exploited these optimizations to undermine the system's security.

Cache attacks on cryptographic algorithms were the first microarchitectural attacks [12, 42, 59]. Osvik et al. [58] showed that an attacker can observe the cache state at the granularity of a cache set using *Prime+Probe*. Yarom et al. [81] proposed *Flush+Reload*, a technique that can observe victim activity at a cache-line granularity. Both *Prime+Probe* and *Flush+Reload* are generic techniques that allow implementing a variety of different attacks, e.g., on cryptographic algorithms [12, 25, 50, 54, 59, 66, 81], web server function calls [83], user input [31, 48, 82], and address layout [24]. *Flush+Reload* requires shared memory between the attacker and the victim. When attacking the last-level cache, *Prime+Probe* requires it to be shared and inclusive. While some Intel processors do not have inclusive last-level caches anymore [80], AMD always focused on non-inclusive or exclusive last-level caches [37]. Without inclusivity and shared memory, these attacks do not apply to AMD CPUs.

With the recent transient-execution attacks, adversaries can directly exfiltrate otherwise inaccessible data on the system [41, 49, 64, 68, 74]. However, AMD's microarchitectures seem to be vulnerable to only a few of them [7, 16]. Consequently, AMD CPUs do not require software mitigations with high performance penalties. Additionally, with the performance improvements of the latest microarchitectures, the share of AMD CPU's used is currently increasing in the cloud [2] and consumer desktops [34].

Since the Bulldozer microarchitecture [5], AMD uses an L1D cache way predictor in their processors. The predictor computes a μTag using an undocumented hash function on the virtual address. This μTag is used to look up the L1D cache way in a prediction table. Hence, the CPU has to compare the cache tag in only one way instead of all possible ways, reducing the power consumption.

In this paper, we present the first attacks on cache way predictors. For

this purpose, we reverse-engineered the undocumented hash function of AMD's L1D cache way predictor in microarchitectures from 2001 up to 2019. We discovered two different hash functions that have been implemented in AMD's way predictors. Knowledge of these functions is the basis of our attack techniques. In the first attack technique, Collide+Probe, we exploit μTag collisions of virtual addresses to monitor the memory accesses of a victim time-sharing the same logical core. Collide+Probe does not require shared memory between the victim and the attacker, unlike *Flush+Reload*, and no knowledge of physical addresses, unlike *Prime+Probe*. In the second attack technique, Load+Reload, we exploit the property that a physical memory location can only reside once in the L1D cache. Thus, accessing the same location with a different virtual address evicts the location from the L1D cache. This allows an attacker to monitor memory accesses on a victim, even if the victim runs on a sibling logical core. Load+Reload is on par with *Flush+Reload* in terms of accuracy and can achieve a higher temporal resolution as it does not invalidate a cache line in the entire cache hierarchy. This allows stealthier attacks that do not induce last-level-cache evictions.

We demonstrate the implications of Collide+Probe and Load+Reload in different attack scenarios. First, we implement a covert channel between two processes with a transmission rate of up to 588.9 kB/s outperforming state-of-the-art covert channels. Second, we use μTag collisions to reduce the entropy of different ASLR implementations. We break kernel ASLR on a fully updated Linux system and demonstrate entropy reduction on user-space applications, the hypervisor, and even on our own address space from sandboxed JavaScript. Furthermore, we successfully recover the secret key using Collide+Probe on an AES T-table implementation. Finally, we use Collide+Probe as a covert channel in a Spectre attack to exfiltrate secret data from the kernel. While we still use a cache-based covert channel, in contrast to previous attacks [41, 44, 51, 70], we do not rely on shared memory between the user application and the kernel. We propose different countermeasures in software and hardware, mitigating Collide+Probe and Load+Reload on current systems and in future designs.

Contributions. The main contributions are as follows:

1. We reverse engineer the L1D cache way predictor of AMD CPUs and provide the addressing functions for virtually all microarchitectures.

2. We uncover the L1D cache way predictor as a source of side-channel leakage and present two new cache-attack techniques, Collide+Probe and Load+Reload.
3. We show that Collide+Probe is on par with *Flush+Reload* and *Prime+Probe* but works in scenarios where other cache attacks fail.
4. We demonstrate and evaluate our attacks in sandboxed JavaScript and virtualized cloud environments.

Responsible Disclosure. We responsibly disclosed our findings to AMD on August 23rd, 2019.

Outline. Section 2 provides background information on CPU caches, cache attacks, way prediction, and simultaneous multithreading (SMT). Section 3 describes the reverse engineering of the way predictor that is necessary for our Collide+Probe and Load+Reload attack techniques outlined in Section 4. In Section 5, we evaluate the attack techniques in different scenarios. Section 6 discusses the interactions between the way predictor and other CPU features. We propose countermeasures in Section 7 and conclude our work in Section 8.

2. Background

In this section, we provide background on CPU caches, cache attacks, high-resolution timing sources, simultaneous multithreading (SMT), and way prediction.

2.1. CPU Caches

CPU caches are a type of memory that is small and fast, that the CPU uses to store copies of data from main memory to hide the latency of memory accesses. Modern CPUs have multiple cache levels, typically three, varying in size and latency: the L1 cache is the smallest and fastest, while the L3 cache, also called the last-level cache, is bigger and slower.

Modern caches are set-associative, *i.e.*, a cache line is stored in a fixed set determined by either its virtual or physical address. The L1 cache typically has 8 ways per set, and the last-level cache has 12 to 20 ways,

depending on the size of the cache. Each line can be stored in any of the ways of a cache set, as determined by the replacement policy. While the replacement policy for the L1 and L2 data cache on Intel is most of the time pseudo least-recently-used (LRU) [1], the replacement policy for the last-level cache (LLC) can differ [78]. Intel CPUs until Sandy Bridge use pseudo least-recently-used (LRU), for newer microarchitectures it is undocumented [78].

The last-level cache is physically indexed and shared across cores of the same CPU. In most Intel implementations, it is also inclusive of L1 and L2, which means that all data in L1 and L2 is also stored in the last-level cache. On AMD Zen processors, the L1D cache is virtually indexed and physically tagged (VIPT). On AMD processors, the last-level cache is a non-inclusive victim cache while on most Intel CPUs it is inclusive. To maintain the inclusiveness property, every line evicted from the last-level cache is also evicted from L1 and L2. The last-level cache, though shared across cores, is also divided into slices. The undocumented hash function on Intel CPUs that maps physical addresses to slices has been reverse-engineered [52].

2.2. Cache Attacks

Cache attacks are based on the timing difference between accessing cached and non-cached memory. They can be leveraged to build side-channel attacks and covert channels. Among cache attacks, access-driven attacks are the most powerful ones, where an attacker monitors its own activity to infer the activity of its victim. More specifically, an attacker detects which cache lines or cache sets the victim has accessed.

Access-driven attacks can further be categorized into two types, depending on whether or not the attacker shares memory with its victim, e.g., using a shared library or memory deduplication. *Flush+Reload* [81], *Evict+Reload* [31] and *Flush+Flush* [30] all rely on shared memory that is also shared in the cache to infer whether the victim accessed a particular cache line. The attacker evicts the shared data either by using the `clflush` instruction (*Flush+Reload* and *Flush+Flush*), or by accessing congruent addresses, *i.e.*, cache lines that belong to the same cache set (*Evict+Reload*). These attacks have a very fine granularity (*i.e.*, a 64-byte memory region), but they are not applicable if shared memory is not available in the corresponding environment. Especially in the cloud, shared memory is usually not available across VMs as memory deduplica-

tion is disabled for security concerns [75]. Irazoqui et al. [37] showed that an attack similar to *Flush+Reload* is also possible in a cross-CPU attack. It exploits that cache invalidations (e.g., from `clflush`) are propagated to all physical processors installed in the same system. When reloading the data, as in *Flush+Reload*, they can distinguish the timing difference between a cache hit in a remote processor and a cache miss, which goes to DRAM.

The second type of access-driven attacks, called *Prime+Probe* [38, 50, 59], does not rely on shared memory and is, thus, applicable to more restrictive environments. As the attacker has no shared cache line with the victim, the `clflush` instruction cannot be used. Thus, the attacker has to access congruent addresses instead (cf. *Evict+Reload*). The granularity of the attack is coarser, *i.e.*, an attacker only obtains information about the accessed cache set. Hence, this attack is more susceptible to noise. In addition to the noise caused by other processes, the replacement policy makes it hard to guarantee that data is actually evicted from a cache set [29].

With the general development to switch from inclusive caches to non-inclusive caches, Intel introduced cache directories. Yan et al. [80] showed that the cache directory is still inclusive, and an attacker can evict a cache directory entry of the victim to invalidate the corresponding cache line. This allows mounting *Prime+Probe* and *Evict+Reload* attacks on the cache directory. They also analyzed whether the same attack works on AMD Piledriver and Zen processors and discovered that it does not, because these processors either do not use a directory or use a directory with high associativity, preventing cross-core eviction either way. Thus, it remains to be answered what types of eviction-based attacks are feasible on AMD processors and on which microarchitectural structures.

2.3. High-resolution Timing

For most cache attacks, the attacker requires a method to measure timing differences in the range of a few CPU cycles. The `rdtsc` instruction provides unprivileged access to a model-specific register returning the current cycle count and is commonly used for cache attacks on Intel CPUs. Using this instruction, an attacker can get timestamps with a resolution between 1 and 3 cycles on modern CPUs. On AMD CPUs, this register has a cycle-accurate resolution until the Zen microarchitecture. Since then, it has a significantly lower resolution as it is only updated every 20

to 35 cycles (cf. Section A). Thus, `rdtsc` is only sufficient if the attacker can repeat the measurement and use the average timing differences over all executions. If an attacker tries to monitor one-time events, the `rdtsc` instruction on AMD cannot directly be used to observe timing differences, which are only a few CPU cycles.

The AMD Ryzen microarchitecture provides the *Actual Performance Frequency Clock Counter* (`APERF` counter) [3] which can be used to improve the accuracy of the timestamp counter. However, it can only be accessed in kernel mode. Although other timing primitives provided by the kernel, such as `get_monotonic_time`, provide nanosecond resolution, they can be more noisy and still not sufficiently accurate to observe timing differences, which are only a few CPU cycles.

Hence, on more recent AMD CPUs, it is necessary to resort to a different method for timing measurements. Lipp et al. [48] showed that *counting threads* can be used on ARM-based devices where unprivileged high-resolution timers are unavailable. Schwarz et al. [66] showed that a counting thread can have a higher resolution than the `rdtsc` instruction on Intel CPUs. A counting thread constantly increments a global variable used as a timestamp without relying on microarchitectural specifics and, thus, can also be used on AMD CPUs.

2.4. Simultaneous Multithreading (SMT)

Simultaneous Multithreading (SMT) allows optimizing the efficiency of superscalar CPUs. SMT enables multiple independent threads to run in parallel on the same physical core sharing the same resources, e.g., execution units and buffers. This allows utilizing the available resources better, increasing the efficiency and throughput of the processor. While on an architectural level, the threads are isolated from each other and cannot access data of other threads, on a microarchitectural level, the same physical resources may be used. Intel introduced SMT as *Hyperthreading* in 2002. AMD introduced 2-way SMT with the Zen microarchitecture in 2017.

Recently, microarchitectural attacks also targeted different shared resources: the TLB [23], store buffer [15], execution ports [8, 13], fill-buffers [64, 68], and load ports [64, 68].

2.5. Way Prediction

To look up a cache line in a set-associative cache, bits in the address determine in which set the cache line is located. With an n-way cache, n possible entries need to be checked for a tag match. To avoid wasting power for n comparisons leading to a single match, Inoue et al. [36] presented way prediction for set-associative caches. Instead of checking all ways of the cache, a way is predicted, and only this entry is checked for a tag match. As only one way is activated, the power consumption is reduced. If the prediction is correct, the access has been completed, and access times similar to a direct-mapped cache are achieved. If the prediction is incorrect, a normal associative check has to be performed.

We only describe AMD's way predictor [6, 22] in more detail in the following section. However, other CPU manufacturers hold patents for cache way prediction as well [56, 63]. CPU's like the Alpha 21264 [40] also implement way prediction to combine the advantages of set-associative caches and the fast access time of a direct-mapped cache.

3. Reverse-engineering AMDs Way Predictor

In this section, we explain how to reverse-engineer the L1D way predictor used in AMD CPUs since the Bulldozer microarchitecture. First, we explain how the AMD L1D way predictor predicts the L1D cache way based on hashed virtual addresses. Second, we reverse-engineer the undocumented hash function used for the way prediction in different microarchitectures. With the knowledge of the hash function and how the L1D way predictor works, we can then build powerful side-channel attacks exploiting AMD's way predictor.

3.1. Way Predictor

Since the AMD Bulldozer microarchitecture, AMD uses a way predictor in the L1 data cache [5]. By predicting the cache way, the CPU only has to compare the cache tag in one way instead of all ways. While this reduces the power consumption of an L1D lookup [6], it may increase the latency in the case of a misprediction.

Every cache line in the L1D cache is tagged with a linear-address-based μTag [6, 22]. This μTag is computed using an undocumented hash function, which takes the virtual address as the input. For every memory load,

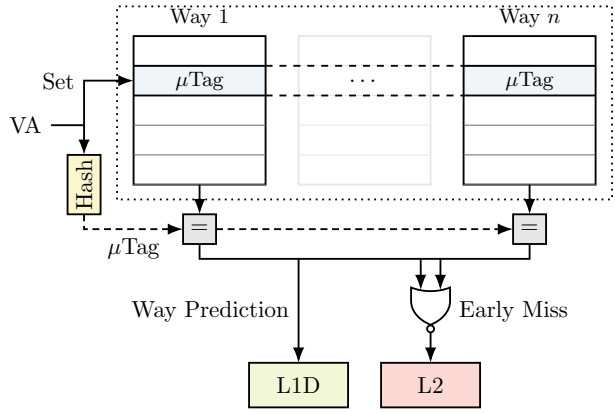


Figure 5.1.: Simplified illustration of AMD's way predictor.

the way predictor predicts the cache way of every memory load based on this μTag. As the virtual address, and thus the μTag, is known before the physical address, the CPU does not have to wait for the TLB lookup. Figure 5.1 illustrates AMD's way predictor. If there is no match for the calculated μTag, an early miss is detected, and a request to L2 issued.

Aliased cache lines can induce performance penalties, *i.e.*, two different *virtual* addresses map to the same *physical* location. VIPT caches with a size lower or equal the number of ways multiplied by the page size behave functionally like PIPT caches. Hence, there are no duplicates for aliased addresses and, thus, in such a case where data is loaded from an aliased address, the load sees an L1D cache miss and thus loads the data from the L2 data cache [6]. If there are multiple memory loads from aliased virtual addresses, they all suffer an L1D cache miss. The reason is that every load updates the μTag and thus ensures that any other aliased address sees an L1D cache miss [6]. In addition, if two different virtual addresses yield the same μTag, accessing one after the other yields a conflict in the μTag table. Thus, an L1D cache miss is suffered, and the data is loaded from the L2 data cache.

3.2. Hash Function

The L1D way predictor computes a hash (μTag) from the virtual address, which is used for the lookup to the way-predictor table. We assume that this undocumented hash function is linear based on the knowledge of other such hash functions, e.g., the cache-slice function of Intel CPUs [52], the

DRAM-mapping function of Intel, ARM, and AMD CPUs [4, 60, 71], or the hash function for indirect branch prediction on Intel CPUs [41]. Moreover, we expect the size of the μTag to be a power of 2, resulting in a linear function.

We rely on μTag collisions to reverse-engineer the hash function. We pick two random virtual addresses that map to the same cache set. If the two addresses have the same μTag, repeatedly accessing them one after the other results in conflicts. As the data is then loaded from the L2 cache, we can either measure an increased access time or observe an increased number in the performance counter for L1 misses, as illustrated in Figure 5.2.

Creating Sets. With the ability to detect conflicts, we can build N sets representing the number of entries in the μTag table. First, we create a pool v of virtual addresses, which all map to the same cache set, *i.e.*, where bits 6 to 11 of the virtual address are the same. We start with one set S_0 containing one random virtual address out of the pool v. For each other randomly-picked address v_x, we measure the access time while alternatively accessing v_x and an address from each set $S_{0...n}$. If we encounter a high access time, we measure conflicts and add v_x to that set. If v_x does not conflict with any existing set, we create a new set S_{n+1} containing v_x.

In our experiments, we recovered 256 sets. Due to measurement errors caused by system noise, there are sets with single entries that can be discarded. Furthermore, to retrieve all sets, we need to make sure to test against virtual addresses where a wide range of bits is set covering the yet unknown bits used by the hash function.

Recovering the Hash Function. Every virtual address, which is in the same set, produces the same hash. To recover the hash function, we need to find which bits in the virtual address are used for the 8 output bits that map to the 256 sets. Due to its linearity, each output bit of the hash function can be expressed as a series of XORs of bits in the virtual address. Hence, we can express the virtual addresses as an over-determined linear equation system in finite field 2, *i.e.*, GF(2). The solutions of the equation system are then linear functions that produce the μTag from the virtual address.

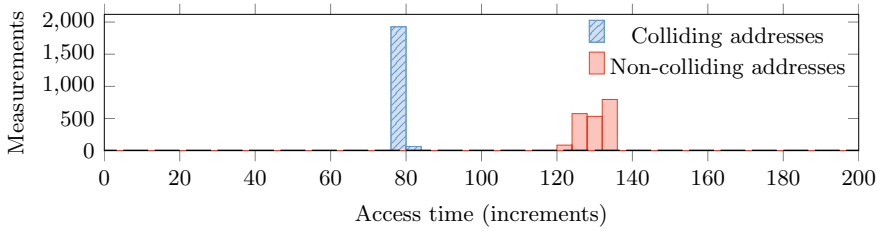


Figure 5.2.: Measured duration of 250 alternating accesses to addresses with and without the same μTag.

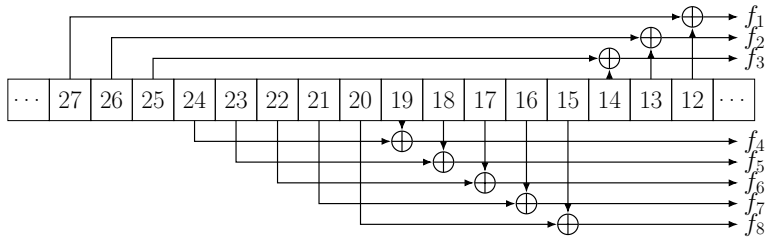
To build the equation system, we use each of the virtual addresses in the 256 sets. For every virtual address, the b bits of the virtual address a are the coefficients, and the bits of the hash function x are the unknown. The right-hand side of the equation y is the same for all addresses in the set. Hence, for every address a in set s, we get an equation of the form $a_{b-1}x_{b-1} \oplus a_{b-2}x_{b-2} \oplus \cdots \oplus a_{12}x_{12} = y_s$.

While the least-significant bits 0-5 define the cache line offset, note that bits 6-11 determine the cache set and are not used for the μTag computation [6]. To solve the equation system, we used the Z3 SMT solver. Every solution vector represents a function which XORs the virtual-address bits that correspond to '1'-bits in the solution vector. The hash function is the set of linearly independent functions, *i.e.*, every linearly independent function yields one bit of the hash function. The order of the bits cannot be recovered. However, this is not relevant, as we are only interested whether addresses collide, not in their numeric μTag value.

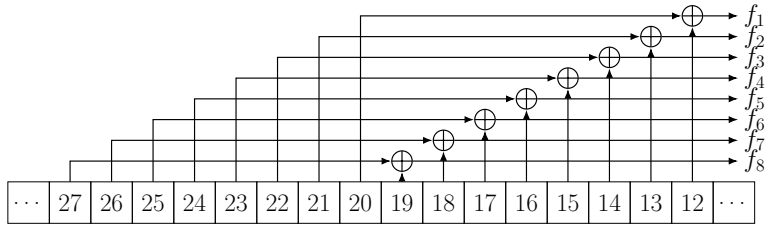
We successfully recovered the undocumented μTag hash function on the AMD Zen, Zen+ and Zen 2 microarchitecture. The function illustrated in Figure 5.3a uses bits 12 to 27 to produce an 8-bit value mapping to one of the 256 sets:

$$h(v) = (v_{12} \oplus v_{27}) \parallel (v_{13} \oplus v_{26}) \parallel (v_{14} \oplus v_{25}) \parallel (v_{15} \oplus v_{20}) \parallel (v_{16} \oplus v_{21}) \parallel (v_{17} \oplus v_{22}) \parallel (v_{18} \oplus v_{23}) \parallel (v_{19} \oplus v_{24})$$

We recovered the same function for various models of the AMD Zen microarchitectures that are listed in Table 5.1. For the Bulldozer microarchitecture (FX-4100), the Piledriver microarchitecture (FX-8350), and the Steamroller microarchitecture (A10-7870K), the hash function uses the same bits but in a different combination Figure 5.3b.



(a) Zen, Zen+, Zen 2



(b) Bulldozer, Piledriver, Steamroller

Figure 5.3.: The recovered hash functions use bits 12 to 27 of the virtual address to compute the μTag.

3.3. Simultaneous Multithreading

As AMD introduced simultaneous multithreading starting with the Zen microarchitecture, the filed patent [22] does not cover any insights on how the way predictor might handle multiple threads. While the way predictor has been used since the Bulldozer microarchitecture [5], parts of the way predictor have only been documented with the release of the Zen microarchitecture [6]. However, the influence of simultaneous multithreading is not mentioned.

Typically, two sibling threads can either share a hardware structure *competitively* with the option to tag entries or by *statically partitioning* them. For instance, on the Zen microarchitecture, execution units, schedulers, or the cache are competitively shared, and the store queue and retire queue are statically partitioned [17]. Although the load queue, as well as the instruction and data TLB, are competitively shared between the threads, the data in these structures can only be accessed by the thread owning it.

Under the assumption that the data structures of the way predictor are competitively shared between threads, one thread could directly influence the sibling thread, enabling cross-thread attacks. We validate this as-

sumption by accessing two addresses with the same μTag on both threads. However, we do not observe collisions, neither by measuring the access time nor in the number of L1 misses. While we reverse-engineered the same mapping function (see Section 3.2) for both threads, the possibility remains that additional per-thread information is used for selecting the data-structure entry, allowing one thread to evict entries of the other.

Hence, we extend the experiment in accessing addresses mapping to all possible μTags on one hardware thread (and all possible cache sets). While we repeatedly accessed one of these addresses on one hardware thread, we measure the number of L1 misses to a single virtual address on the sibling thread. However, we are not able to observe any collisions and, thus, conclude that either individual structures are used per thread or that they are shared but tagged for each thread. The only exceptions are aliased loads as the hardware updates the μTag in the aliased way (see Section 3.1).

In another experiment, we measure access times of two virtual addresses that are mapped to the same physical address. As documented [6], loads to an aliased address see an L1D cache miss and, thus, load the data from the L2 data cache. While we verified this behavior, we additionally observed that this is also the case if the other thread performs the other load. Hence, the structure used is searched by the sibling thread, suggesting a competitively shared structure that is tagged with the hardware threads.

4. Using the Way Predictor for Side Channels

In this section, we present two novel side channels that leverage AMD's L1D cache way predictor. With Collide+Probe, we monitor memory accesses of a victim's process without requiring the knowledge of physical addresses. With Load+Reload, while relying on shared memory similar to *Flush+Reload*, we can monitor memory accesses of a victim's process running on the sibling hardware thread without invalidating the targeted cache line from the entire cache hierarchy.

4.1. Collide+Probe

Collide+Probe is a new cache side channel exploiting μTag collisions in AMD's L1D cache way predictor. As described in Section 3, the way

predictor uses virtual-address-based μTags to predict the L1D cache way. If an address is accessed, the μTag is computed, and the way-predictor entry for this μTag is updated. If a subsequent access to a different address with the same μTag is performed, a μTag collision occurs, and the data has to be loaded from the L2D cache, increasing the access time. With Collide+Probe, we exploit this timing difference to monitor accesses to such colliding addresses.

Threat Model. For this attack, we assume that the attacker has unprivileged native code execution on the target machine and runs on the same logical CPU core as the victim. Furthermore, the attacker can force the execution of the victim's code, e.g., via a function call in a library or a system call.

Setup. The attacker first chooses a virtual address v of the victim that should be monitored for accesses. This can be an arbitrary valid address in the victim's address space. There are no constraints in choosing the address. The attacker can then compute the μTag μ_v of the target address using the hash function from Section 3.2. We assume that ASLR is either not active or has already been broken (cf. Section 5.2). However, although with ASLR, the actual virtual address used in the victim's process are typically unknown to the attacker, it is still possible to mount an attack. Instead of choosing a virtual address, the attacker initially performs a cache template attack [31] to detect which of 256 possible μTags should be monitored. Similar to *Prime+Probe* [58], where the attacker monitors the activity of cache sets, the attacker monitors μTag collisions while triggering the victim.

Attack. To mount a Collide+Probe attack, the attacker selects a virtual address v' in its own address space that yields the same μTag $\mu_{v'}$ as the target address v, *i.e.*, $\mu_v = \mu_{v'}$. As there are only 256 different μTags, this can easily be done by randomly choosing addresses until the chosen address has the same μTag. Moreover, both v and v' have to be in the same cache set. However, this is easily satisfiable, as the cache set is determined by bits 6-11 of the virtual address. The attack consists of 3 phases performed repeatedly:

Phase 1: Collide. In the first phase, the attacker accesses the precomputed address v' and, thus, updates the way predictor. The way

predictor associates the cache line of v' with its μTag $\mu_{v'}$ and subsequent memory accesses with the same μTag are predicted to be in the same cache way. Since the victim's address v has the same μTag ($\mu_v = \mu_{v'}$), the μTag of that cache line is marked invalid and the data is effectively inaccessible from the L1D cache.

Phase 2: Scheduling the victim. In the second phase, the victim is scheduled to perform its operations. If the victim does not access the monitored address v, the way predictor remains in the same state as set up by the attacker. Thus, the attacker's data is still accessible from the L1D cache. However, if the victim performs an access to the monitored address v, the way predictor is updated again causing the attacker's data to be inaccessible from L1D.

Phase 3: Probe. In the third and last phase of the attack, the attacker measures the access time to the pre-computed address v'. If the victim has not accessed the monitored address v, the data of the pre-computed address v' is still accessible from the L1D cache and the way prediction is correct. Thus, the measured access time is fast. If the victim has accessed the monitored address v and thus changed the state of the way predictor, the attacker suffers an L1D cache miss when accessing v', as the prediction is now incorrect. The data of the pre-computed address v' is loaded from the L2 cache and, thus, the measured access time is slow. By distinguishing between these cases, the attacker can deduce whether the victim has accessed the targeted data.

Listing 5.1 shows an implementation of the Collide+Probe attack where the colliding address `colliding_address` is computed beforehand. The code closely follows the three attack phases. First, the colliding address is accessed. Then, the victim is scheduled, illustrated by the `run_victim` function. Afterwards, the access time to the same address is measured where the `get_time` function is implemented using a timing source discussed in Section 2.3. The measured access time allows the attacker to distinguish between an L1D cache hit and an L2-cache hit and, thus, deduce if the victim has accessed the targeted address. As other accesses with the same cache set influence the measurements, the attacker can repeat the experiment to average out the measured noise.

Comparison to Other Cache Attacks. Finally, we want to discuss the advantages and disadvantages of the Collide+Probe attack in comparison to other cache side-channel attacks. In contrast to *Prime+Probe*, no

```
1 access(colliding_address);
2 run_victim();
3 size_t begin = get_time();
4 access(colliding_address);
5 size_t end = get_time() − begin;
6 if ((end − begin) > THRESHOLD) report_event();
```

Listing 5.1: Implementation of the Collide+Probe attack

knowledge of physical addresses is required as the way predictor uses the virtual address to compute μTags. Thus, with native code execution, an attacker can find addresses corresponding to a specific μTag without any effort. Another advantage of Collide+Probe over *Prime+Probe* is that a single memory load is enough to guarantee that a subsequent load with the same μTag is served from the L2 cache. With *Prime+Probe*, multiple loads are required to ensure that the target address is evicted from the cache. In modern *Prime+Probe* attacks, the last-level cache is targeted [38, 48, 50, 67], and knowledge of physical addresses is required to compute both the cache set and cache slice [52]. While Collide+Probe requires knowledge of virtual addresses, they are typically easier to get than physical addresses. In contrast to *Flush+Reload*, Collide+Probe does neither require any specific instructions like `clflush` nor shared memory between the victim and the attacker. A disadvantage is that distinguishing L1D from L2 hits in Collide+Probe requires a timing primitive with higher precision than required to distinguish cache hits from misses in *Flush+Reload*.

4.2. Load+Reload

Load+Reload exploits the way predictor's behavior for aliased address, *i.e.*, virtual addresses mapping to the same physical address. When accessing data through a virtual-address alias, the data is always requested from the L2 cache instead of the L1D cache [6]. By monitoring the performance counter for L1 misses, we also observe this behavior across hardware threads. Consequently, this allows one thread to evict shared data used by the sibling thread with a single load. Although the requested data is stored in the L1D cache, it remains inaccessible for the other thread and, thus, introduces a timing difference when it is accessed.

Threat Model. For this attack, we assume that the attacker has unprivileged native code execution on the target machine. The attacker and victim run simultaneously on the same physical but different logical CPU thread. The attack target is a memory location with virtual address v shared between the attacker and victim, e.g., a shared library.

Attack. Load+Reload exploits the timing difference when accessing a virtual-address alias v' to build a cross-thread attack on shared memory. The attack consists of 3 phases:

Phase 1: Load. In contrast to *Flush+Reload*, where the targeted address v is flushed from the cache hierarchy, Load+Reload loads an address v' with the same physical tag as v in the first phase. Thereby, it renders the cache line containing v inaccessible from the L1D cache for the sibling thread.

Phase 2: Scheduling the victim. In the second phase, the victim process is scheduled. If the victim process accesses the targeted cache line with address v, it sees an L1D cache miss. As a result, it loads the data from the L2 cache, invalidating the attacker's cache line with address v' in the process.

Phase 3: Reload. In the third phase, the attacker measures the access time to the address v'. If the victim process has accessed the cache line with address v, the attacker observes an L1D cache miss and loads the data from the L2 cache, resulting in a higher access time. Otherwise, if the victim has not accessed the cache line with address v, it is still accessible in the L1D cache for the attacker and, thus, a lower access time is measured. By distinguishing between both cases, the attacker can deduce whether the victim has accessed the address v.

Comparison with *Flush+Reload*. While *Flush+Reload* invalidates a cache line from the entire cache hierarchy, Load+Reload only *evicts* the data for the sibling thread from the L1D. Thus, Load+Reload is limited to cross-thread scenarios, while *Flush+Reload* is applicable to cross-core scenarios too.

5. Case Studies

To demonstrate the impact of the side channel introduced by the μTag, we implement different attack scenarios. In Section 5.1, we implement a covert channel between two processes with a transmission rate of up to 588.9 kB/s outperforming state-of-the-art covert channels. In Section 5.2, we break kernel ASLR, demonstrate how user-space ASLR can be weakened, and reduce the ASLR entropy of the hypervisor in a virtual-machine setting. In Section 5.3, we use Collide+Probe as a covert channel to extract secret data from the kernel in a Spectre attack. In Section 5.4, we recover secret keys in AES T-table implementations.

Timing Measurement. As explained in Section 2.3, we cannot rely on the `rdtsc` instruction for high-resolution timestamps on AMD CPUs since the Zen microarchitecture. As we use recent AMD CPUs for our evaluation, we use a counting thread (cf. Section 2.3) running on the sibling logical CPU core for most of our experiments if applicable. In other cases, e.g., a covert channel scenario, the counting thread runs on a different physical CPU core.

Environment. We evaluate our attacks on different environments listed in Table 5.1, with CPUs from K8 (released 2013) to Zen 2 (released in 2019). We have reverse-engineered 2 unique hash functions, as described in Section 3. One is the same for all Zen microarchitectures, and the other is the same for all previous microarchitectures with a way predictor.

5.1. Covert Channel

A covert channel is a communication channel between two parties that are not allowed to communicate with each other. Such a covert channel can be established by leveraging a side channel. The μTag used by AMD's L1D way prediction enables a covert channel for two processes accessing addresses with the same μTag.

For the most simplistic form of the covert channel, two processes agree on a μTag and a cache set (*i.e.*, the least-significant 12 bits of the virtual addresses are the same). This μTag is used for sending and receiving data by inducing and measuring cache misses.

Table 5.1.: Tested CPUs with their microarchitecture (μ-arch.) and whether they have a way predictor (WP).

Setup	**CPU**	**μ-arch.**	WP
Lab	AMD Athlon 64 X2 3800+	K8	✗
Lab	AMD Turion II Neo N40L	K10	✗
Lab	AMD Phenom II X6 1055T	K10	✗
Lab	AMD E-450	Bobcat	✗
Lab	AMD Athlon 5350	Jaguar	✗
Lab	AMD FX-4100	Bulldozer	✓
Lab	AMD FX-8350	Piledriver	✓
Lab	AMD A10-7870K	Steamroller	✓
Lab	AMD Ryzen Threadripper 1920X	Zen	✓
Lab	AMD Ryzen Threadripper 1950X	Zen	✓
Lab	AMD Ryzen Threadripper 1700X	Zen	✓
Lab	AMD Ryzen Threadripper 2970WX	Zen+	✓
Lab	AMD Ryzen 7 3700X	Zen 2	✓
Cloud	AMD EPYC 7401p	Zen	✓
Cloud	AMD EPYC 7571	Zen	✓

In the initialization phase, both parties allocate their own page. The sender chooses a virtual address v_S, and the receiver chooses a virtual address v_R that fulfills the aforementioned requirements, *i.e.*, v_S and v_R are in the same cache set and yield the same μTag. The μTag can simply be computed using the reverse-engineered hash function of Section 3.

To encode a 1-bit to transmit, the sender accesses address v_S. To transmit a 0-bit, the sender does not access address v_S. The receiving end decodes the transmitted information by measuring the access time when loading address v_R. If the sender has accessed address v_S to transmit a 1, the collision caused by the same μTag of v_S and v_R results in a slow access time for the receiver. If the sender has not accessed address v_S, no collision caused the address v_R to be evicted from L1D and, thus, the access time is fast. This timing difference allows the receiver to decode the transmitted bit.

Different cache-based covert channels use the same side channel to transmit multiple bits at once. For instance, different cache lines [30, 48] or different cache sets [48, 53] are used to encode one bit of information on its own. We extended the described μTag covert channel to transmit

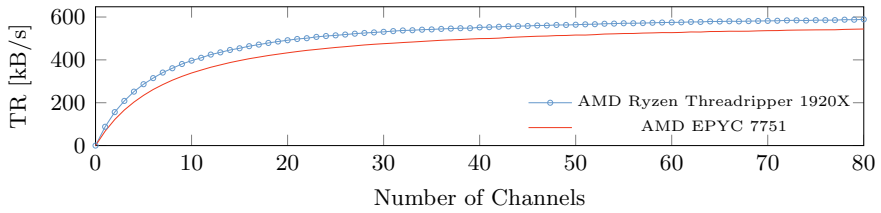


Figure 5.4.: Mean transmission rate of the covert channels using multiple parallel channels on different CPUs.

multiple bits in parallel by utilizing multiple cache sets. Instead of decoding the transmitted bit based on the timing difference of one address, we use two addresses in two cache sets for every bit we transmit: One to represent a 1-bit and the other to represent the 0-bit. As the L1D has 64 cache sets, we can transmit up to 32 bit in parallel without reusing cache sets.

Performance Evaluation. We evaluated the transmission and error rate of our covert channel in a local setting and a cloud setting by sending and receiving a randomly generated data blob. We achieved a maximum transmission rate of 588.9 kB/s ($\sigma_{\bar{x}} = 0.544$, $n = 1000$) using 80 channels in parallel on the AMD Ryzen Threadripper 1920X. On the AMD EPYC 7571 in the Amazon EC2 cloud, we achieved a maximum transmission rate of 544.0 kB/s ($\sigma_{\bar{x}} = 0.548$, $n = 1000$) also using 80 channels. In contrast, L1 *Prime+Probe* achieved a transmission rate of 400 kB/s [59] and *Flush+Flush* a transmission rate of 496 kB/s [30]. As illustrated in Figure 5.4, the mean transmission rate increases with the number of bits sent in parallel. However, the error rate increases drastically when transmitting more than 64 bits in parallel, as illustrated in Figure 5.6. As the number of available different cache sets for our channel is exhausted for our covert channel, sending more bits in parallel would reuse already used sets. This increases the chance of wrong measurements and, thus, the error rate.

Error Correction. As accesses to unrelated addresses with the same μTag as our covert channel introduce noise in our measurements, an attacker can use error correction to achieve better transmission. Using hamming codes [33], we introduce n additional parity bits allowing us to detect and correct wrongly measured bits of a packet with a size of $2^n - 1$

bits. For our covert channel, we implemented different Hamming codes $H(m, n)$ that encode n bits by adding $m-n$ parity bits. The receiving end of the covert channel computes the parity bits from the received data and compares it with the received parity bits. Naturally, they only differ if a transmission error occurred. The erroneous bit position can be computed, and the bit error corrected by flipping the bit. This allows to detect up to 2-bit errors and correct one-bit errors for a single transmission.

We evaluated different hamming codes on an AMD Ryzen Threadripper 1920X, as illustrated in Figure 5.7 in Section B. When sending data through 60 parallel channels, the $H(7, 4)$ code reduces the error rate to 0.14 % ($\sigma_{\bar{x}} = 0.08$, $n = 1000$), whereas the $H(15, 11)$ code achieves an error rate of 0.16 % ($\sigma_{\bar{x}} = 0.08$, $n = 1000$). While the $H(7, 4)$ code is slightly more robust [33], the $H(15, 11)$ code achieves a better transmission rate of 452.2 kB/s ($\sigma_{\bar{x}} = 7.79$, $n = 1000$).

More robust protocols have been used in cache-based covert channels in the past [48, 53] to achieve error-free communication. While these techniques can be applied to our covert channel as well, we leave it up to future work.

Limitations. As we are not able to observe μTag collisions between two processes running on sibling threads on one physical core, our covert channel is limited to processes running on the same logical core.

5.2. Breaking ASLR and KASLR

To exploit a memory corruption vulnerability, an attacker often requires knowledge of the location of specific data in memory. With *address space layout randomization* (ASLR), a basic memory protection mechanism has been developed that randomizes the locations of memory sections to impede the exploitation of these bugs. ASLR is not only applied to user-space applications but also implemented in the kernel (KASLR), randomizing the offsets of code, data, and modules on every boot.

In this section, we exploit the relation between virtual addresses and μTags to reduce the entropy of ASLR in different scenarios. With Collide+Probe, we can determine the μTags accessed by the victim, e.g., the kernel or the browser, and use the reverse-engineered mapping functions (Section 3.2) to infer bits of the addresses. We show an additional attack on heap ASLR in Section C.

Table 5.2.: Evaluation of the ASLR experiments

Target	Entropy	Bits Reduced	Success Rate	Timing Source	Time
Linux Kernel	9	7	98.5%	thread	0.51 ms (σ = 12.12 µs)
User Process	13	13	88.9%	thread	1.94 s (σ = 1.76 s)
Virt. Manager	28	16	90.0%	`rdtsc`	2.88 s (σ = 3.16 s)
Virt. Module	18	8	98.9%	`rdtsc`	0.14 s (σ = 1.74 ms)
Mozilla Firefox	28	15	98.0%	web worker	2.33 s (σ = 0.03 s)
Google Chrome	28	15	86.1%	web worker	2.90 s (σ = 0.25 s)
Chrome V8	28	15	100.0%	`rdtsc`	1.14 s (σ = 0.03 s)

5.2.1. Kernel

On modern Linux systems, the position of the kernel text segment is randomized inside the 1 GB area from `0xffff ffff 8000 0000` - `0xffff ffff c000 0000` [39, 46]. As the kernel image is mapped using 2 MB pages, it can only be mapped in 512 different locations, resulting in 9 bit of entropy [65].

Global variables are stored in the `.bss` and `.data` sections of the kernel image. Since 2 MB physical pages are used, the 21 lower address bits of a global variable are identical to the lower 21 bits of the offset within the the kernel image section. Typically, the kernel image is public and does not differ among users with the same operating system. With the knowledge of the μTag from the address of a global variable, one can compute the address bits 21 to 27 using the hash function of AMD's L1D cache way predictor.

To defeat KASLR using Collide+Probe, the attacker needs to know the offset of a global variable within the kernel image that is accessed by the kernel on a user-triggerable event, e.g., a system call or an interrupt. While not many system calls access global variables, we found that the `SYS_time` system call returns the value of the global second counter `obj.xtime_sec`. Using Collide+Probe, the attacker accesses an address v' with a specific μTag $\mu_{v'}$ and schedules the system call, which accesses the global variable with address v and μTag μ_v. Upon returning from the kernel, the attacker probes the μTag $\mu_{v'}$ using address v'. On a conflict, the attacker infers that the address v' has the same μTag, *i.e.*, $t = \mu_{v'} = \mu_v$. Otherwise, the attacker chooses another address v' with a different μTag $\mu_{v'}$ and repeats the process. As the μTag bits t_0 to t_7 are known, the address bits v_{20} to v_{27} can be computed from address bits v_{12} to v_{19} based on the way predictor's hash functions (Section 3.2). Following this approach, we can compute address bits 21 to 27 of the

global variable. As we know the offset of the global variable inside the kernel image, we can also recover the start address of the kernel image mapping, leaving only bits 28 and 29 unknown. As the kernel is only randomized once per boot, the reduction to only 4 address possibilities gives an attacker a significant advantage.

For the evaluation, we tested 10 different randomization offsets on a Linux 4.15.0-58 kernel with an AMD Ryzen Threadripper 1920X processor. We ran our experiment 1000 times for each randomization offset. With a success rate of 98.5 %, we were able to reduce the entropy of KASLR on average in 0.51 ms ($\sigma = 12.12\,\mu s, n = 10\,000$).

While there are several microarchitectural KASLR breaks, this is to the best of our knowledge the first which reportedly works on AMD and not only on Intel CPUs. Hund et al. [35] measured differences in the runtime of page faults when repeatedly accessing either valid or invalid kernel addresses on Intel CPUs. Barresi et al. [11] exploited page deduplication to break ASLR: a copy-on-write pagefault only occurs for the page with the correctly guessed address. Gruss et al. [28] exploited runtime differences in the `prefetch` instruction on Intel CPUs to detect mapped kernel pages. Jang et al. [39] showed that the difference in access time to valid and invalid kernel addresses can be measured when suppressing exceptions with Intel TSX. Evtyushkin et al. [21] exploited the branch-target buffer on Intel CPUs to gain information on mapped pages. Schwarz et al. [65] showed that the store-to-load forwarding logic on Intel CPUs is missing a permission check which allows to detect whether any virtual address is valid. Canella et al. [15] exploited that recent stores can be leaked from the store buffer on vulnerable Intel CPUs, allowing to detect valid kernel addresses.

5.2.2. Hypervisor

The *Kernel-based Virtual Machine* (KVM) is a virtualization module that allows the Linux kernel to act as a hypervisor to run multiple, isolated environments in parallel called virtual machines or guests. Virtual machines can communicate with the hypervisor using hypercalls with the privileged `vmcall` instruction. In the past, collisions in the branch target buffer (BTB) have been used to break hypervisor ASLR [21, 77].

In this scenario, we leak the base address of the KVM kernel module from a guest virtual machine. We issue hypercalls with invalid call num-

bers and monitor, which μTags have been accessed using Collide+Probe. In our evaluation, we identified two cache sets enabling us to weaken ASLR of the `kvm` and the `kvm_amd` module with a success rate of 98.8 % and an average runtime of 0.14 s ($\sigma = 1.74\,\text{ms}, n = 1000$). We verified our results by comparing the leaked address bits with the symbol table (`/proc/kallsyms`).

Another target is the user-space virtualization manager, e.g., QEMU. Guest operating systems can interact with virtualization managers through various methods, e.g., the `out` instruction. Likewise to the previously described hypercall method, a guest virtual machine can use this method to trigger the managing user process to interact with the guest memory from its own address space. By using Collide+Probe in this scenario, we were able to reduce the ASLR entropy by 16 bits with a success rate of 90.0 % with an average run time of 2.88 s ($\sigma = 3.16\,\text{s}, n = 1000$).

5.2.3. JavaScript

In this section, we show that Collide+Probe is not only restricted to native environments. We use Collide+Probe to break ASLR from JavaScript within Chrome and Firefox. As the JavaScript standard does not define a way to retrieve any address information, side channels in browsers have been used in the past [57], also to break ASLR, simplifying browser exploitation [24, 65].

The idea of our ASLR break is similar to the approach of reverse-engineering the way predictor's mapping function, as described in Section 3.2. First, we allocate a large chunk of memory as a JavaScript typed array. If the requested array length is big enough, the execution engine allocates it using `mmap`, placing the array at the beginning of a memory page [29, 69]. This allows using the indices within the array as virtual addresses with an additional constant offset. By accessing pairs of addresses, we can find μTag collisions allowing us to build an equation system where the only unknown bits are the bits of the address where the start of the array is located. As the equation system is very small, an attacker can trivially solve it in JavaScript.

However, to distinguish between colliding and non-colliding addresses, we require a high-precision timer in JavaScript. While the `performance.now()` function only returns rounded results for security reasons [9, 14], we leverage an alternative timing source [24, 69]. For

our evaluation, we used the technique of a counting thread constantly incrementing a shared variable [24, 48, 69, 79].

We tested our proof-of-concept in both the Chrome 76.0.3809 and Firefox 68.0.2 web browsers as well as the Chrome V8 standalone engine. In Firefox, we are able to reduce the entropy by 15 bits with a success rate of 98 % and an average run time of 2.33 s ($\sigma = 0.03\,\text{s}, n = 1000$). With Chrome, we can correctly reduce the bits with a success rate of 86.1 % and an average run time of 2.90 s ($\sigma = 0.25\,\text{s}, n = 1000$). As the JavaScript standard does not provide any functionality to retrieve the addresses used by variables, we extended the capabilities of the Chrome V8 engine to verify our results. We introduced several custom JavaScript functions, including one that returned the virtual address of an array. This provided us with the ground truth to verify that our proof-of-concept recovered the address bits correctly. Inside the extended Chrome V8 engine, we were able to recover the address bits with a success rate of 100 % and an average run time of 1.14 s ($\sigma = 0.03\,\text{s}, n = 1000$).

5.3. Leaking Kernel Memory

In this section, we combine Spectre with Collide+Probe to leak kernel memory without the requirement of shared memory. While some Spectre-type attacks use AVX [70] or port contention [13], most attacks use the cache as a covert channel to encode secrets [16, 41]. During transient execution, the kernel caches a user-space address based on a secret. By monitoring the presence of said address in the cache, the attacker can deduce the leaked value.

As AMD CPU's are not vulnerable to Meltdown [49], stronger kernel isolation [27] is not enforced on modern operating systems, leaving the kernel mapped in user space. However, with SMAP enabled, the processor never loads an address into the cache if the translation triggers a SMAP violation, *i.e.*, the kernel tries to access a user-space address [7]. Thus, an attacker has to find a vulnerable indirect branch that can access user-space memory. We lift this restriction by using Collide+Probe as a cache-based covert channel to infer secret values accessed by the kernel. With Collide+Probe, we can observe μTag collisions based on the secret value that is leaked and, thus, remove the requirement of shared memory, *i.e.*, user memory that is directly accessible to the kernel.

To evaluate Collide+Probe as a covert channel for a Spectre-type attack,

we implement a custom kernel module containing a Spectre-PHT gadget as illustrated as follows:

```
1 if (index < bounds) { a = LUT[data[index] * 4096]; }
```

The execution of the presented code snippet can be triggered with an `ioctl` command that allows the user to control the *index* variable as it is passed as an argument. First, we mistrain the branch predictor by repeatedly providing an index that is in bounds, letting the processor follow the branch to access a fixed kernel-memory location. Then, we access an address that collides with the kernel address accessed based on a possible byte-value located at `data[index]`. By providing an out-of-bounds index, the processor now speculatively accesses a memory location based on the secret data located at the out-of-bounds index. Using Collide+Probe, we can now detect if the kernel has accessed the address based on the assumed secret byte value. By repeating this step for each of the 256 possible byte values, we can deduce the actual byte as we observe μTag conflicts. As we cannot ensure that the processor always misspeculates when providing the out-of-bounds index, we run this attack multiple times for each byte we want to leak.

We successfully leaked a secret string using Collide+Probe as a covert channel on an AMD Ryzen Threadripper 1920X. With our unoptimized version, we are able to leak the secret bytes with a success rate of 99.5 % ($\sigma_{\bar{x}} = 0.19, n = 100$) and a leakage rate of 0.66 B/s ($\sigma_{\bar{x}} = 0.00043, n = 100$). While we leak full byte values in our proof-of-concept, other gadgets could allow to leak bit-wise, reducing the overhead of measuring every possible byte value significantly. In addition, the parameters for the number of mistrainings or the necessary repetitions of the attack to leak a byte can be further tweaked to match the processor under attack. To utilize this side channel, the attacker requires the knowledge of the address of the kernel-memory that is accessed by the gadget. Thus, on systems with active kernel ASLR, the attacker first needs to defeat it. However, as described in Section 5.2, the attacker can use Collide+Probe to derandomize the kernel as well.

5.4. Attacking AES T-Tables

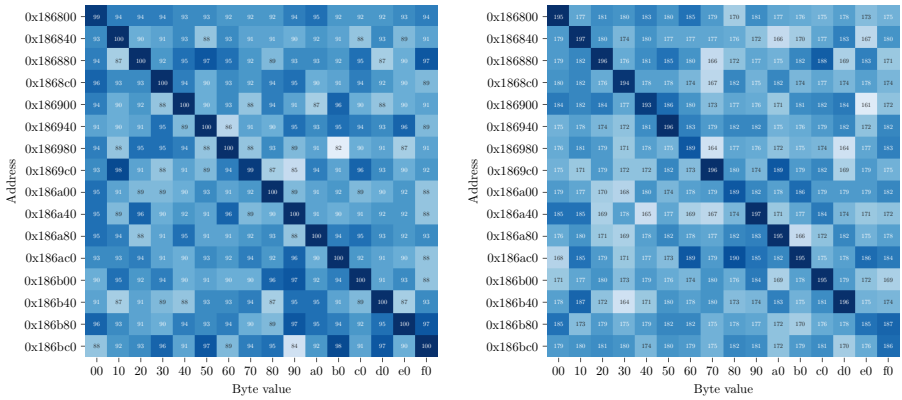
In this section, we show an attack on an AES [19] T-table implementation. While cache attacks have already been demonstrated against T-table implementations [30, 31, 48, 58, 72] and appropriate countermeasures, e.g.,

bit-sliced implementations [43, 62], have been presented, they serve as a good example to demonstrate the applicability of the side channel and allow to compare it against other existing cache side-channels. Furthermore, AES T-tables are still sometimes used in practice. While some implementations fall back to T-table implementations [20] if the AES-NI instruction extension [32] is not available, others only offer T-table-based implementations [45, 55]. For evaluation purposes, we used the T-table implementation of OpenSSL version 1.1.1c.

In this implementation, the *SubBytes*, *ShiftRows*, and *MixColumns* steps of the AES round transformation are replaced by look-ups to 4 pre-computed T-tables T_0, ..., T_3. As the *MixColumns* operation is omitted in the last round, an additional T-table T_4 is necessary. Each table contains 256 4-byte words, requiring 1 kB of memory.

In our proof-of-concept, we mount the first-round attack by Osvik et al. [58]. Let k_i denote the initial key bytes, p_i the plaintext bytes and $x_i = p_i \oplus k_i$ for $i = 0, \ldots, 15$ the initial state of AES. The initial state bytes are used to select elements of the pre-computed T-tables for the following round. An attacker who controls the plaintext byte p_i and monitors which entries of the T-table are accessed can deduce the key byte $k_i = s_i \oplus p_i$. However, with a cache-line size of 64 B, it is only possible to derive the upper 4 bit of k_i if the T-tables are properly aligned in memory. With second-round and last-round attacks [58, 73] or disaligned T-tables [72], the key space can be reduced further.

Figure 5.5 shows the results of a Collide+Probe and a Load+Reload attack on the AMD Ryzen Threadripper 1920X on the first key byte. As the first key byte is set to zero, the diagonal shows a higher number of cache hits than the other parts of the table. We repeated every experiment 1000 times. With Collide+Probe, we can successfully recover with a probability of 100 % ($\sigma_{\bar{x}} = 0$) the upper 4 bits of each k_i with 168 867 ($\sigma_{\bar{x}} = 719$) encryptions per byte in 0.07 s ($\sigma_{\bar{x}} = 0.0003$). With Load+Reload, we require 367 731 ($\sigma_{\bar{x}} = 82388$) encryptions and an average runtime of 0.53 s ($\sigma_{\bar{x}} = 0.11$) to recover 99.0 % ($\sigma_{\bar{x}} = 0.0058$) of the key bits. Using *Prime+Probe* on the L1 cache, we can successfully recover 99.7 % ($\sigma_{\bar{x}} = 0.01$) of the key bits with 450 406 encryptions ($\sigma_{\bar{x}} = 1129$) in 1.23 s ($\sigma_{\bar{x}} = 0.003$).



(a) Collide+Probe (b) Load+Reload

Figure 5.5.: Cache access pattern with Collide+Probe and Load+Reload on the first key byte.

6. Discussion

While the official documentation of the way prediction feature does not explain how it interacts with other processor features, we discuss the interactions with instruction caches, transient execution, and hypervisors.

Instruction Caches. The patent [22] describes that AMD's way predictor can be used for both data and instruction cache. However, AMD only documents a way predictor for the L1D cache [6] and not for the L1I cache.

Transient Execution. Speculative execution is a crucial optimization in modern processors. When the CPU encounters a branch, instead of waiting for the branch condition, the CPU guesses the outcome and continues the execution in a transient state. If the speculation was correct, the executed instructions are committed. Otherwise, they are discarded. Similarly, CPUs employ out-of-order execution to transiently execute instructions ahead of time as soon as their dependencies are fulfilled. On an exception, the transiently executed instructions following the exception are simply discarded, but leave traces in the microarchitectural state [16]. We investigated the possibility that AMD Zen processors use the data from the predicted way without waiting for the physical tag returned by the TLB. However, we were not able to produce any such results.

Hypervisor. AMD does not document any interactions of the way predictor with virtualization. As we have shown in our experiments (cf. Section 5.2), the way predictor does not distinguish between virtual machines and hypervisors. The way predictor uses the virtual address without any tagging, regardless whether it is a guest or host virtual address.

7. Countermeasures

In this section, we discuss mitigations to the presented attacks on AMD's way predictor. We first discuss hardware-only mitigations, followed by mitigations requiring hardware and software changes, as well as a software-only solution.

Temporarily Disable Way Predictor. One solution lies in designing the processor in a way that allows temporarily disabling the way predictor temporarily. Alves et al. [10] evaluated the performance impact penalty of instruction replays caused by mispredictions. By dynamically disabling way prediction, they observe a higher performance than with standard way prediction. Dynamically disabling way prediction can also be used to prevent attacks by disabling it if too many mispredictions within a defined time window are detected. If an adversary tries to exploit the way predictor or if the current legitimate workload provokes too many conflicts, the processor deactivates the way predictor and falls back to comparing the tags from all ways. However, it is unknown whether AMD processors support this in hardware, and there is no documented operating system interface to it.

Keyed Hash Function. The currently used mapping functions (Section 3) rely solely on bits of the virtual address. This allows an attacker to reverse-engineer the used function once and easily find colliding virtual addresses resulting in the same μTag. By keying the mapping function with an additional process- or context-dependent secret input, a reverse-engineered hash function is only valid for the attacker process. ScatterCache [76] and CEASAR-S [61] are novel cache designs preventing cache attacks by introducing a similar keyed mapping function for skewed-associative caches. Hence, we expect that such methods are also effective when used for the way predictor. Moreover, the key can be

updated regularly, e.g., when returning from the kernel, and, thus, not remain the same over the execution time of the program.

State Flushing. With Collide+Probe, an attacker cannot monitor memory accesses of a victim running on a sibling thread. However, μTag collisions can still be observed after context switches or transitions between kernel and user mode. To mitigate Collide+Probe, the state of the way predictor can be cleared when switching to another user-space application or returning from the kernel. Every subsequent memory access yields a misprediction and is thus served from the L2 data cache. This yields the same result as invalidating the L1 data cache, which is currently a required mitigation technique against Foreshadow [74] and MDS attacks [15, 64, 68]. However, we expect it to be more power-efficient than flushing the L1D. To mitigate Spectre attacks [41, 44, 51], it is already necessary to invalidate branch predictors upon context switches [16]. As invalidating predictors and the L1D cache on Intel has been implemented through CPU microcode updates, introducing an MSR to invalidate the way predictor might be possible on AMD as well.

Uniformly-distributed Collisions. While the previously described countermeasures rely on either microcode updates or hardware modifications, we also propose an entirely software-based mitigation. Our attack on an optimized AES T-table implementation in Section 5.4 relies on the fact that an attacker can observe the key-dependent look-ups to the T-tables. We propose to map such secret data n times, such that the data is accessible via n different virtual addresses, which all have a different μTag. When accessing the data, a random address is chosen out of the n possible addresses. The attacker cannot learn which T-table has been accessed by monitoring the accessed μTags, as a uniform distribution over all possibilities will be observed. This technique is not restricted to T-table implementations but can be applied to virtually any secret-dependent memory access within an application. With dynamic software diversity [18], diversified replicas of program parts are generated automatically to thwart cache-side channel attacks.

8. Conclusion

The key takeaway of this paper is that AMD's cache way predictors leak secret information. To understand the implementation details, we reverse engineered AMD's L1D cache way predictor, leading to two novel side-channel attack techniques. First, Collide+Probe allows monitoring memory accesses on the current logical core without the knowledge of physical addresses or shared memory. Second, Load+Reload obtains accurate memory-access traces of applications co-located on the same physical core.

We evaluated our new attack techniques in different scenarios. We established a high-speed covert channel and utilized it in a Spectre attack to leak secret data from the kernel. Furthermore, we reduced the entropy of different ASLR implementations from native code and sandboxed JavaScript. Finally, we recovered a key from a vulnerable AES implementation.

Our attacks demonstrate that AMD's design is vulnerable to side-channel attacks. However, we propose countermeasures in software and hardware, allowing to secure existing implementations and future designs of way predictors.

Acknowledgments

We thank our anonymous reviewers for their comments and suggestions that helped improving the paper. The project was supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria, and Carinthia. It was also supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work also benefited from the support of the project ANR-19-CE39-0007 MIAOUS of the French National Research Agency (ANR). Additional funding was provided by generous gifts from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Andreas Abel and Jan Reineke. "Measurement-based Modeling of the Cache Replacement Policy". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013.

[2] Advanced Micro Devices Inc. *2nd Gen AMD EPYC Processors Set New Standard for the Modern Datacenter with Record-Breaking Performance and Significant TCO Savings*. 2019.

[3] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual*. 2017.

[4] Advanced Micro Devices Inc. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*. 2013.

[5] Advanced Micro Devices Inc. *Software Optimization Guide for AMD Family 15h Processors*. Jan. 2014.

[6] Advanced Micro Devices Inc. *Software Optimization Guide for AMD Family 17h Processors*. 2017.

[7] Advanced Micro Devices Inc. *Software Techniques for Managing Speculation on AMD Processors*. Revison 7.10.18. 2018.

[8] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit". In: *S&P*. 2018.

[9] Alex Christensen. *Reduce resolution of performance.now.* 2015. URL: `https://bugs.webkit.org/show_bug.cgi?id=146531`.

[10] Ricardo Alves, Stefanos Kaxiras, and David Black-Schaffer. "Dynamically disabling way-prediction to reduce instruction replay". In: *International Conference on Computer Design (ICCD)*. 2018.

[11] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. "CAIN: Silently Breaking ASLR in the Cloud". In: *WOOT*. 2015.

[12] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. Tech. rep. 2005. URL: `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[13] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. "SMoTherSpectre: exploiting speculative execution through port contention". In: *CCS*. 2019.

[14] Boris Zbarsky. *Reduce resolution of performance.now.* 2015. URL: `https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab`.

[15] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. "Fallout: Leaking Data on Meltdown-resistant CPUs". In: *CCS*. 2019.

[16] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *USENIX Security Symposium*. Extended classification tree and PoCs at https://transient.fail/. 2019.

[17] Mike Clark. "A new x86 core architecture for the next generation of computing". In: *IEEE Hot Chips Symposium (HCS)*. 2016.

[18] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity". In: *NDSS*. 2015.

[19] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. 2013.

[20] Helder Eijs. *PyCryptodome: A self-contained cryptographic library for Python*. 2018. URL: `https://www.pycryptodome.org`.

[21] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR". In: *MICRO*. 2016.

[22] W. Shen Gene and S. Craig Nelson. *MicroTLB and micro tag for reducing power in a processor*. US Patent 7,117,290 B2. Oct. 2006.

[23] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *USENIX Security Symposium*. 2018.

[24] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU". In: *NDSS*. 2017.

[25] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. "Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme". In: *CHES*. 2016.

[26] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel computing* (1996).

[27] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. “KASLR is Dead: Long Live KASLR”. In: *ESSoS*. 2017.

[28] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *CCS*. 2016.

[29] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA*. 2016.

[30] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA*. 2016.

[31] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

[32] Shay Gueron. *Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01*. 2012.

[33] Richard W Hamming. “Error detecting and error correcting codes”. In: *The Bell system technical journal* (1950).

[34] Joel Hruska. *AMD Gains Market Share in Desktop and Laptop, Slips in Servers*. 2019. URL: `https://www.extremetech.com/computing/291032-amd`.

[35] Ralf Hund, Carsten Willems, and Thorsten Holz. “Practical Timing Side Channel Attacks against Kernel Space ASLR”. In: *S&P*. 2013.

[36] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. “Way-predicting set-associative cache for high performance and low energy consumption”. In: *Symposium on Low Power Electronics and Design*. 1999.

[37] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cross processor cache attacks”. In: *AsiaCCS*. 2016.

[38] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES”. In: *S&P*. 2015.

[39] Yeongjin Jang, Sangho Lee, and Taesoo Kim. “Breaking Kernel Address Space Layout Randomization with Intel TSX”. In: *CCS*. 2016.

[40] Richard E Kessler. “The alpha 21264 microprocessor”. In: *IEEE Micro* (1999).

[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P*. 2019.

[42] Paul C. Kocher. "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems". In: *CRYPTO*. 1996.

[43] Robert Könighofer. "A Fast and Cache-Timing Resistant Implementation of the AES". In: *CT-RSA*. 2008.

[44] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer". In: *WOOT*. 2018.

[45] Marcin Krzyzanowski. *CryptoSwift: Growing collection of standard and secure cryptographic algorithms implemented in Swift*. 2019. URL: `https://cryptoswift.io`.

[46] Linux. *Complete virtual memory map with 4-level page tables*. 2019. URL: `https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt`.

[47] Linux. *Linux Kernel 5.0 Process (x86)*. 2019. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/kernel/process.c`.

[48] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security Symposium*. 2016.

[49] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security Symposium*. 2018.

[50] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *S&P*. 2015.

[51] G. Maisuradze and C. Rossow. "ret2spec: Speculative Execution Using Return Stack Buffers". In: *CCS*. 2018.

[52] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "Reverse Engineering Intel Complex Addressing Using Performance Counters". In: *RAID*. 2015.

[53] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and

Kay Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: *NDSS*. 2017.

[54] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “Cachezoom: How SGX amplifies the power of cache attacks”. In: *CHES*. 2017.

[55] Richard Moore. *pyaes: Pure-Python implementation of AES block-cipher and common modes of operation*. 2017. URL: `https://github.com/ricmoo/pyaes`.

[56] Louis-Marie Vincent Mouton, Nicolas Jean Phillippe Huot, Gilles Eric Grandou, and Stephane Eric Sebastian Brochier. *Cache accessing using a micro TAG*. US Patent 8,151,055. 2012.

[57] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS*. 2015.

[58] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: the Case of AES”. In: *CT-RSA*. 2006.

[59] Colin Percival. “Cache missing for fun and profit”. In: *BSDCan*. 2005.

[60] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security Symposium*. 2016.

[61] Moinuddin K Qureshi. “New attacks and defense for encrypted-address cache”. In: *ISCA*. 2019.

[62] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. “Bitslice Implementation of AES”. In: *Cryptology and Network Security (CANS)*. 2006.

[63] David J Sager and Glenn J Hinton. *Way-predicting cache memory*. US Patent 6,425,055. 2002.

[64] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “RIDL: Rogue In-flight Data Load”. In: *S&P*. 2019.

[65] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. “Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs”. In: *arXiv:1905.05725* (2019).

[66] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks ”. In: *DIMVA*. 2017.

[67] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard.

"KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks". In: *NDSS*. 2018.

[68] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *CCS*. 2019.

[69] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *FC*. 2017.

[70] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network". In: *ESORICS*. 2019.

[71] Mark Seaborn. *How physical addresses map to rows and banks in DRAM*. 2015. URL: `http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html`.

[72] Raphael Spreitzer and Thomas Plos. "Cache-Access Pattern Attack on Disaligned AES T-Tables". In: *COSADE*. 2013.

[73] Junko Takahashi, Toshinori Fukunaga, Kazumaro Aoki, and Hitoshi Fuji. "Highly accurate key extraction method for access-driven cache attacks using correlation coefficient". In: *ACISP*. 2013.

[74] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security Symposium*. 2018.

[75] VMWare. *Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing (2080735)*. 2018. URL: `https://kb.vmware.com/s/article/2080735`.

[76] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization". In: *USENIX Security Symposium*. 2019.

[77] Felix Wilhelm. *PoC for breaking hypervisor ASLR using branch target buffer collisions*. 2016. URL: `https://github.com/felixwilhelm/mario_baslr`.

[78] Henry Wong. *Intel Ivy Bridge Cache Replacement Policy*. 2013. URL: `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`.

[79] John C Wray. “An analysis of covert timing channels”. In: *Journal of Computer Security* 1.3-4 (1992), pp. 219–232.

[80] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. “Attack directories, not caches: Side channel attacks in a non-inclusive world”. In: *S&P*. 2019.

[81] Yuval Yarom and Katrina Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security Symposium*. 2014.

[82] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. “Return-oriented flush-reload side channels on arm and their implications for android devices”. In: *CCS*. 2016.

[83] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Cross-Tenant Side-Channel Attacks in PaaS Clouds”. In: *CCS*. 2014.

Appendix

A. RDTSC Resolution

We measure the resolution of the `rdtsc` instruction using the following experimental setup. We assume that the timestamp counter (TSC) is updated in a fixed interval. This assumption is based on the documentation in the manual that the timestamp counter is independent of the CPU frequency [6]. Hence, there is a modulus x and a constant C, such that TSC $\mod x \equiv C$ iff x is the TSC increment. We can easily find this x with brute-force, *i.e.*, trying all different x until we find an x, which always results in the same value C. Table 5.3 shows a `rdtsc` increments for the CPUs we tested.

B. Covert Channel Error Rate

Figure 5.6 illustrates the error rate of the covert channel described in Section 5.1. The error rate increases drastically when transmitting more than 64 bits in parallel. Thus, we evaluated different hamming codes on an AMD Ryzen Threadripper 1920X (Figure 5.7).

Table 5.3.: `rdtsc` increments on various CPUs.

Setup	CPU	μ-arch.	Increment
Lab	AMD Athlon 64 X2 3800+	K8	1
Lab	AMD Turion II Neo N40L	K10	1
Lab	AMD Phenom II X6 1055T	K10	1
Lab	AMD E-450	Bobcat	1
Lab	AMD Athlon 5350	Jaguar	1
Lab	AMD FX-4100	Bulldozer	1
Lab	AMD FX-8350	Piledriver	1
Lab	AMD A10-7870K	Steamroller	1
Lab	AMD Ryzen Threadripper 1920X	Zen	35
Lab	AMD Ryzen Threadripper 1950X	Zen	34
Lab	AMD Ryzen Threadripper 1700X	Zen	34
Lab	AMD Ryzen Threadripper 2970WX	Zen+	30
Lab	AMD Ryzen 7 3700X	Zen 2	36
Cloud	AMD EPYC 7401p	Zen	20
Cloud	AMD EPYC 7571	Zen	22

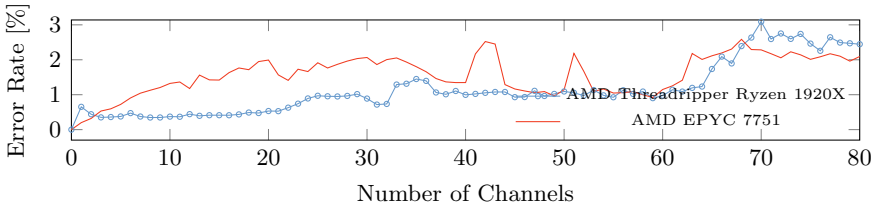


Figure 5.6.: Error rate of the covert channel.

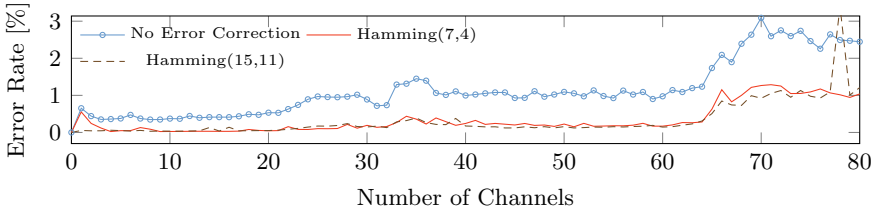


Figure 5.7.: Error rate of the covert channel with and without error correction using different Hamming codes.

C. Userspace ASLR

Linux also uses ASLR for user processes by default. However, randomizing the code section requires compiler support for position-independent code. The heap memory region is of particular interest because it is located just after the code section with an offset of up to 32 MB [47]. User programs use 4 kB pages, giving an effective 13-bit entropy for the start of the brk-based heap memory.

It is possible to fully break heap ASLR through the use of μTags. An attack requires an interface to the victim application that incurs a victim access to data on the heap. We evaluated the ASLR break using a client-server scenario in a toy application, where the attacker is the malicious client. The attacker repeatedly sends benign requests until it is distinguishable which tag is being accessed by the victim. This already reduces the ASLR entropy by 8 bits because it reveals a linear combination of the address bits. It is also possible to recover all address bits up to bit 27 by using the μTags of multiple pages and solving the resulting equation system.

Again, a limitation is that the attack is susceptible to noise. Too many accesses while processing the attacker's request negatively impact the measurements such that the attacker will always observe a cache miss. In our experiments, we were not able to mount the attack using a socket-based interface. Hence, attacking other user-space applications that rely on a more complex interface, e.g., using D-Bus, is currently not practical. However, future work may refine our techniques to also mount attacks in more noisy scenarios. For our evaluation, we targeted a shared-memory-based API for high-speed transmission without system calls [26] provided by the victim application. We were able to recover 13 bits with an average success rate of 88.9 % in 1.94 s ($\sigma = 1.76\,\text{s}, n = 1000$).

6

Meltdown: Reading Kernel Memory from User Space

Publication Data

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security Symposium*. 2018

Contributions

Main author.

Meltdown: Reading Kernel Memory from User Space

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2], Werner Haas[2], Anders Fogh[3], Jann Horn[4], Stefan Mangard[1], Paul Kocher[5], Daniel Genkin[6], 9, Yuval Yarom[7], Mike Hamburg[8]

[1]Graz University of Technology, [2]Cyberus Technology GmbH, [3]G-Data Advanced Analytics, [4]Google Project Zero, [5]Independent (www.paulkocher.com), [6]University of Michigan, [7]University of Adelaide & Data61, [8]Rambus, Cryptography Research Division

Abstract

The security of computer systems fundamentally relies on memory isolation, e.g., kernel address ranges are marked as non-accessible and are protected from user access. In this paper, we present Meltdown. Meltdown exploits side effects of out-of-order execution on modern processors to read arbitrary kernel-memory locations including personal data and passwords. Out-of-order execution is an indispensable performance feature and present in a wide range of modern processors. The attack is independent of the operating system, and it does not rely on any software vulnerabilities. Meltdown breaks all security guarantees provided by address space isolation as well as paravirtualized environments and, thus, every security mechanism building upon this foundation. On affected systems, Meltdown enables an adversary to read memory of other processes or virtual machines in the cloud without any permissions or privileges, affecting millions of customers and virtually every user of a personal computer. We show that the KAISER defense mechanism for KASLR has the important (but inadvertent) side effect of impeding Meltdown. We stress that KAISER must be deployed immediately to prevent large-scale exploitation of this severe information leakage.

1. Introduction

A central security feature of today's operating systems is memory isolation. Operating systems ensure that user programs cannot access each other's memory or kernel memory. This isolation is a cornerstone of our

[9]Work was partially done while the author was affiliated to University of Pennsylvania and University of Maryland.

computing environments and allows running multiple applications at the same time on personal devices or executing processes of multiple users on a single machine in the cloud.

On modern processors, the isolation between the kernel and user processes is typically realized by a supervisor bit of the processor that defines whether a memory page of the kernel can be accessed or not. The basic idea is that this bit can only be set when entering kernel code and it is cleared when switching to user processes. This hardware feature allows operating systems to map the kernel into the address space of every process and to have very efficient transitions from the user process to the kernel, e.g., for interrupt handling. Consequently, in practice, there is no change of the memory mapping when switching from a user process to the kernel.

In this work, we present Meltdown[10]. Meltdown is a novel attack that allows overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executes on, including all physical memory mapped in the kernel region. Meltdown does not exploit any software vulnerability, *i.e.*, it works on all major operating systems. Instead, Meltdown exploits side-channel information available on most modern processors, e.g., modern Intel microarchitectures since 2010 and potentially on other CPUs of other vendors.

While side-channel attacks typically require very specific knowledge about the target application and are tailored to only leak information about its secrets, Meltdown allows an adversary who can run code on the vulnerable processor to obtain a dump of the entire kernel address space, including any mapped physical memory. The root cause of the simplicity and strength of Meltdown are side effects caused by *out-of-order execution*.

Out-of-order execution is an important performance feature of today's processors in order to overcome latencies of busy execution units, e.g., a memory fetch unit needs to wait for data arrival from memory. Instead of stalling the execution, modern processors run operations *out-of-order i.e.*, they look ahead and schedule subsequent operations to idle execution units of the core. However, such operations often have unwanted side-

[10]Using the practice of responsible disclosure, disjoint groups of authors of this paper provided preliminary versions of our results to partially overlapping groups of CPU vendors and other affected companies. In coordination with industry, the authors participated in an embargo of the results. Meltdown is documented under CVE-2017-5754.

effects, e.g., timing differences [23, 55, 63] can leak information from both sequential and out-of-order execution.

From a security perspective, one observation is particularly significant: vulnerable out-of-order CPUs allow an unprivileged process to load data from a privileged (kernel or physical) address into a temporary CPU register. Moreover, the CPU even performs further computations based on this register value, e.g., access to an array based on the register value. By simply discarding the results of the memory lookups (e.g., the modified register states), if it turns out that an instruction should not have been executed, the processor ensures correct program execution. Hence, on the architectural level (e.g., the abstract definition of how the processor should perform computations) no security problem arises.

However, we observed that out-of-order memory lookups influence the cache, which in turn can be detected through the cache side channel. As a result, an attacker can dump the entire kernel memory by reading privileged memory in an out-of-order execution stream, and transmit the data from this elusive state via a microarchitectural covert channel (e.g., *Flush+Reload*) to the outside world. On the receiving end of the covert channel, the register value is reconstructed. Hence, on the microarchitectural level (e.g., the actual hardware implementation), there is an exploitable security problem.

Meltdown breaks all security guarantees provided by the CPU's memory isolation capabilities. We evaluated the attack on modern desktop machines and laptops, as well as servers in the cloud. Meltdown allows an unprivileged process to read data mapped in the kernel address space, including the entire physical memory on Linux, Android and OS X, and a large fraction of the physical memory on Windows. This may include the physical memory of other processes, the kernel, and in the case of kernel-sharing sandbox solutions (e.g., Docker, LXC) or Xen in paravirtualization mode, the memory of the kernel (or hypervisor), and other co-located instances. While the performance heavily depends on the specific machine, e.g., processor speed, TLB and cache sizes, and DRAM speed, we can dump arbitrary kernel and physical memory with 3.2 KB/s to 503 KB/s. Hence, an enormous number of systems are affected.

The countermeasure KAISER [20], developed initially to prevent side-channel attacks targeting KASLR, inadvertently protects against Meltdown as well. Our evaluation shows that KAISER prevents Meltdown to a large extent. Consequently, we stress that it is of utmost importance

to deploy KAISER on all operating systems immediately. Fortunately, during a responsible disclosure window, the three major operating systems (Windows, Linux, and OS X) implemented variants of KAISER and recently rolled out these patches.

Meltdown is distinct from the Spectre Attacks [40] in several ways, notably that Spectre requires tailoring to the victim process's software environment, but applies more broadly to CPUs and is not mitigated by KAISER.

Contributions. The contributions of this work are:

1. We describe out-of-order execution as a new, extremely powerful, software-based side channel.

2. We show how out-of-order execution can be combined with a microarchitectural covert channel to transfer the data from an elusive state to a receiver on the outside.

3. We present an end-to-end attack combining out-of-order execution with exception handlers or TSX, to read arbitrary physical memory without any permissions or privileges, on laptops, desktop machines, mobile phones and on public cloud machines.

4. We evaluate the performance of Meltdown and the effects of KAISER on it.

Outline. The remainder of this paper is structured as follows: In Section 2, we describe the fundamental problem which is introduced with out-of-order execution. In Section 3, we provide a toy example illustrating the side channel Meltdown exploits. In Section 4, we describe the building blocks of Meltdown. We present the full attack in Section 5. In Section 6, we evaluate the performance of the Meltdown attack on several different systems and discuss its limitations. In Section 7, we discuss the effects of the software-based KAISER countermeasure and propose solutions in hardware. In Section 8, we discuss related work and conclude our work in Section 9.

2. Background

In this section, we provide background on out-of-order execution, address translation, and cache attacks.

2.1. Out-of-order execution

Out-of-order execution is an optimization technique that allows maximizing the utilization of all execution units of a CPU core as exhaustive as possible. Instead of processing instructions strictly in the sequential program order, the CPU executes them as soon as all required resources are available. While the execution unit of the current operation is occupied, other execution units can run ahead. Hence, instructions can be run in parallel as long as their results follow the architectural definition.

In practice, CPUs supporting out-of-order execution allow running operations *speculatively* to the extent that the processor's out-of-order logic processes instructions before the CPU is certain that the instruction will be needed and committed. In this paper, we refer to speculative execution in a more restricted meaning, where it refers to an instruction sequence following a branch, and use the term out-of-order execution to refer to any way of getting an operation executed before the processor has committed the results of all prior instructions.

In 1967, Tomasulo [61] developed an algorithm that enabled dynamic scheduling of instructions to allow out-of-order execution. Tomasulo [61] introduced a unified reservation station that allows a CPU to use a data value as it has been computed instead of storing it in a register and re-reading it. The reservation station renames registers to allow instructions that operate on the same physical registers to use the last logical one to solve read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) hazards. Furthermore, the reservation unit connects all execution units via a common data bus (CDB). If an operand is not available, the reservation unit can listen on the CDB until it is available and then directly begin the execution of the instruction.

On the Intel architecture, the pipeline consists of the front-end, the execution engine (back-end) and the memory subsystem [33]. x86 instructions are fetched by the front-end from memory and decoded to micro-operations (μOPs) which are continuously sent to the execution engine. Out-of-order execution is implemented within the execution engine as illustrated in Figure 6.1. The *Reorder Buffer* is responsible for register allo-

cation, register renaming and retiring. Additionally, other optimizations like move elimination or the recognition of zeroing idioms are directly handled by the reorder buffer. The μOPs are forwarded to the *Unified Reservation Station* (Scheduler) that queues the operations on exit ports that are connected to *Execution Units*. Each execution unit can perform different tasks like ALU operations, AES operations, address generation units (AGU) or memory loads and stores. AGUs, as well as load and store execution units, are directly connected to the memory subsystem to process its requests.

Since CPUs usually do not run linear instruction streams, they have branch prediction units that are used to obtain an educated guess of which instruction is executed next. Branch predictors try to determine which direction of a branch is taken before its condition is actually evaluated. Instructions that lie on that path and do not have any dependencies can be executed in advance and their results immediately used if the prediction was correct. If the prediction was incorrect, the reorder buffer allows to rollback to a sane state by clearing the reorder buffer and re-initializing the unified reservation station.

There are various approaches to predict a branch: With static branch prediction [28], the outcome is predicted solely based on the instruction itself. Dynamic branch prediction [8] gathers statistics at run-time to predict the outcome. One-level branch prediction uses a 1-bit or 2-bit counter to record the last outcome of a branch [45]. Modern processors often use two-level adaptive predictors [64] with a history of the last n outcomes, allowing to predict regularly recurring patterns. More recently, ideas to use neural branch prediction [38, 60, 62] have been picked up and integrated into CPU architectures [9].

2.2. Address Spaces

To isolate processes from each other, CPUs support virtual address spaces where virtual addresses are translated to physical addresses. A virtual address space is divided into a set of pages that can be individually mapped to physical memory through a multi-level page translation table. The translation tables define the actual virtual to physical mapping and also protection properties that are used to enforce privilege checks, such as readable, writable, executable and user-accessible. The currently used translation table is held in a special CPU register. On each context switch, the operating system updates this register with the next process'

translation table address in order to implement per-process virtual address spaces. Because of that, each process can only reference data that belongs to its virtual address space. Each virtual address space itself is split into a user and a kernel part. While the user address space can be accessed by the running application, the kernel address space can only be accessed if the CPU is running in privileged mode. This is enforced by the operating system disabling the user-accessible property of the corresponding translation tables. The kernel address space does not only have memory mapped for the kernel's own usage, but it also needs to perform operations on user pages, e.g., filling them with data. Consequently, the entire physical memory is typically mapped in the kernel. On Linux and OS X, this is done via a direct-physical map, *i.e.*, the entire physical memory is directly mapped to a pre-defined virtual address (cf. Figure 6.2).

Instead of a direct-physical map, Windows maintains a multiple so-called *paged pools*, *non-paged pools*, and the *system cache*. These pools are virtual memory regions in the kernel address space mapping physical pages to virtual addresses which are either required to remain in the memory (*non-paged pool*) or can be removed from the memory because a copy is already stored on the disk (*paged pool*). The *system cache* further contains mappings of all file-backed pages. Combined, these memory pools will typically map a large fraction of the physical memory into the kernel address space of every process.

The exploitation of memory corruption bugs often requires knowledge of addresses of specific data. In order to impede such attacks, address space layout randomization (ASLR) has been introduced as well as non-executable stacks and stack canaries. To protect the kernel, kernel ASLR (KASLR) randomizes the offsets where drivers are located on every boot, making attacks harder as they now require to guess the location of kernel data structures. However, side-channel attacks allow to detect the exact location of kernel data structures [21, 29, 37] or derandomize ASLR in JavaScript [16]. A combination of a software bug and the knowledge of these addresses can lead to privileged code execution.

2.3. Cache Attacks

In order to speed-up memory accesses and address translation, the CPU contains small memory buffers, called caches, that store frequently used data. CPU caches hide slow memory access latencies by buffering fre-

quently used data in smaller and faster internal memory. Modern CPUs have multiple levels of caches that are either private per core or shared among them. Address space translation tables are also stored in memory and, thus, also cached in the regular caches.

Cache side-channel attacks exploit timing differences that are introduced by the caches. Different cache attack techniques have been proposed and demonstrated in the past, including *Evict+Time* [55], *Prime+Probe* [55, 56], and *Flush+Reload* [63]. *Flush+Reload* attacks work on a single cache line granularity. These attacks exploit the shared, inclusive last-level cache. An attacker frequently flushes a targeted memory location using the `clflush` instruction. By measuring the time it takes to reload the data, the attacker determines whether data was loaded into the cache by another process in the meantime. The *Flush+Reload* attack has been used for attacks on various computations, e.g., cryptographic algorithms [4, 36, 63], web server function calls [65], user input [23, 47, 58], and kernel addressing information [21].

A special use case of a side-channel attack is a covert channel. Here the attacker controls both, the part that induces the side effect, and the part that measures the side effect. This can be used to leak information from one security domain to another, while bypassing any boundaries existing on the architectural level or above. Both *Prime+Probe* and *Flush+Reload* have been used in high-performance covert channels [22, 48, 52].

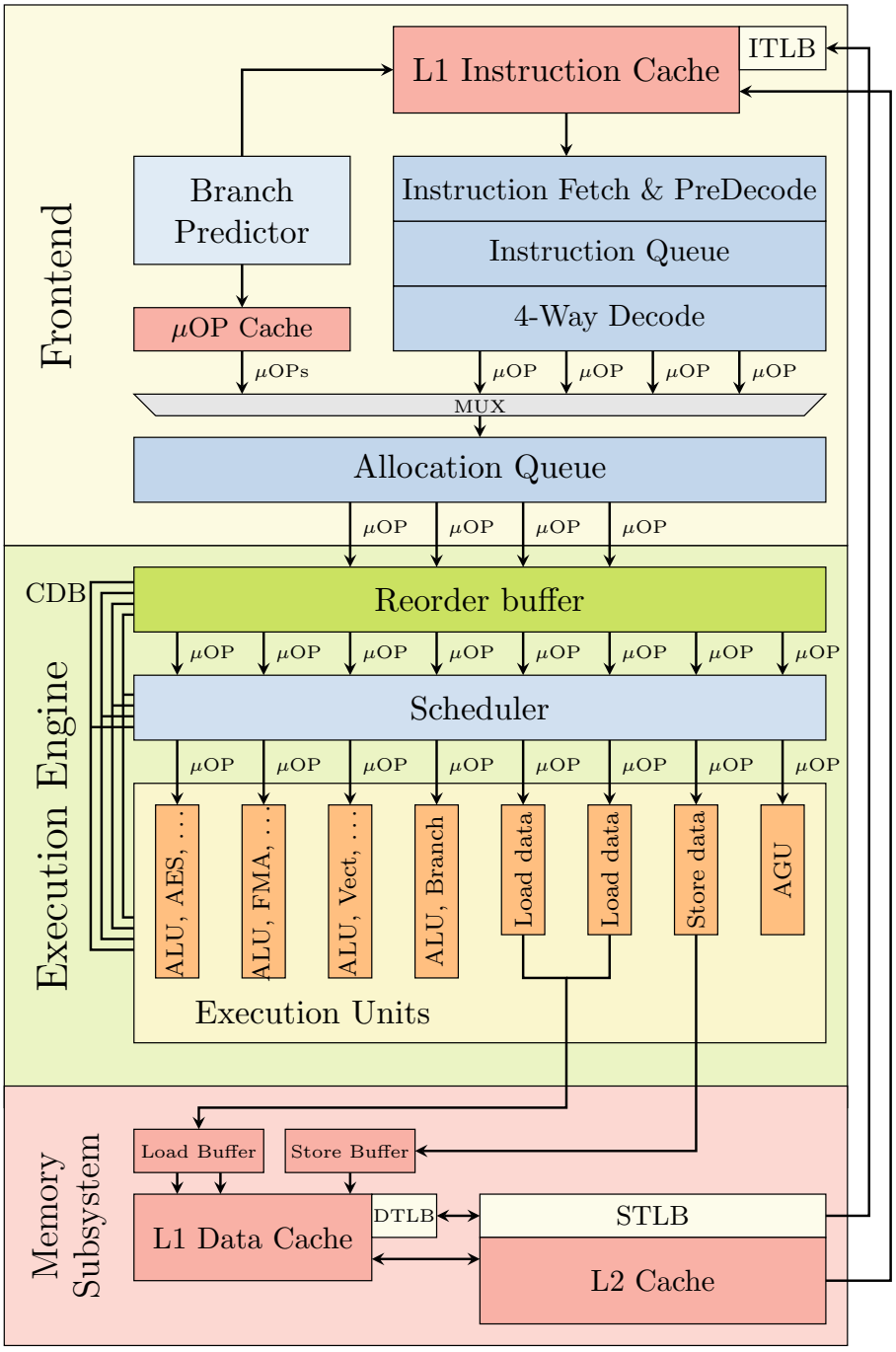


Figure 6.1.: Simplified illustration of a single core of the Intel's Skylake microarchitecture. Instructions are decoded into μOPs and executed out-of-order in the execution engine by individual execution units.

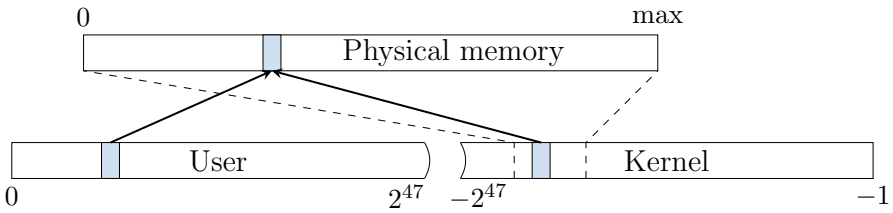


Figure 6.2.: The physical memory is directly mapped in the kernel at a certain offset. A physical address (blue) which is mapped accessible to the user space is also mapped in the kernel space through the direct mapping.

3. A Toy Example

In this section, we start with a toy example, *i.e.*, a simple code snippet, to illustrate that out-of-order execution can change the microarchitectural state in a way that leaks information. However, despite its simplicity, it is used as a basis for Section 4 and Section 5, where we show how this change in state can be exploited for an attack.

Listing 6.1 shows a simple code snippet first raising an (unhandled) exception and then accessing an array. The property of an exception is that the control flow does not continue with the code after the exception, but jumps to an exception handler in the operating system. Regardless of whether this exception is raised due to a memory access, e.g., by accessing an invalid address, or due to any other CPU exception, e.g., a division by zero, the control flow continues in the kernel and not with the next user space instruction.

Thus, our toy example cannot access the array in theory, as the exception immediately traps to the kernel and terminates the application. However, due to the out-of-order execution, the CPU might have already executed the following instructions as there is no dependency on the instruction triggering the exception. This is illustrated in Figure 6.3. Due to the exception, the instructions executed out of order are not retired and, thus, never have architectural effects.

Although the instructions executed out of order do not have any visible architectural effect on registers or memory, they have microarchitectural side effects. During the out-of-order execution, the referenced memory is fetched into a register and also stored in the cache. If the out-of-order execution has to be discarded, the register and memory contents are never committed. Nevertheless, the cached memory contents are kept in the cache. We can leverage a microarchitectural side-channel attack such as *Flush+Reload* [63], which detects whether a specific memory location is cached, to make this microarchitectural state visible. Other side channels

```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

Listing 6.1: A toy example to illustrate side-effects of out-of-order execution.

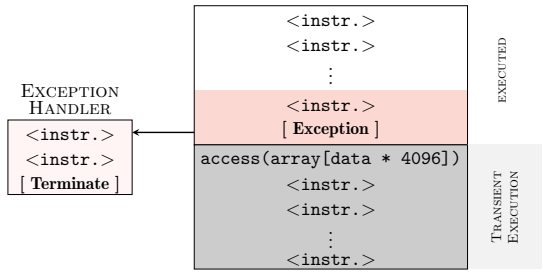


Figure 6.3.: If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed. Due to out-of-order execution, the subsequent instructions may already have been partially executed, but not retired. However, architectural effects of the execution are discarded.

can also detect whether a specific memory location is cached, including *Prime+Probe* [48, 52, 55], *Evict+Reload* [47], or *Flush+Flush* [22]. As *Flush+Reload* is the most accurate known cache side channel and is simple to implement, we do not consider any other side channel for this example.

Based on the value of `data` in this example, a different part of the cache is accessed when executing the memory access out of order. As `data` is multiplied by 4096, data accesses to `probe_array` are scattered over the array with a distance of 4 KB (assuming an 1 B data type for `probe_array`). Thus, there is an injective mapping from the value of `data` to a memory page, *i.e.*, different values for data never result in an access to the same page. Consequently, if a cache line of a page is cached, we know the value of `data`. The spreading over pages eliminates false positives due to the prefetcher, as the prefetcher cannot access data across page boundaries [33].

Figure 6.4 shows the result of a *Flush+Reload* measurement iterating over all pages, after executing the out-of-order snippet with `data = 84`. Although the array access should not have happened due to the exception, we can clearly see that the index which would have been accessed is cached. Iterating over all pages (e.g., in the exception handler) shows only a cache hit for page 84 This shows that even instructions which are never actually executed, change the microarchitectural state of the CPU. Section 4 modifies this toy example not to read a value but to leak an inaccessible secret.

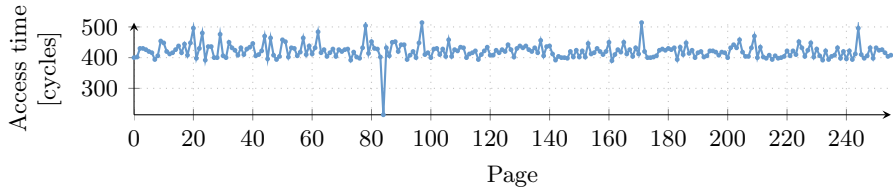


Figure 6.4.: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

4. Building Blocks of the Attack

The toy example in Section 3 illustrated that side-effects of out-of-order execution can modify the microarchitectural state to leak information. While the code snippet reveals the data value passed to a cache-side channel, we want to show how this technique can be leveraged to leak otherwise inaccessible secrets. In this section, we want to generalize and discuss the necessary building blocks to exploit out-of-order execution for an attack.

The adversary targets a secret value that is kept somewhere in physical memory. Note that register contents are also stored in memory upon context switches, *i.e.*, they are also stored in physical memory. As described in Section 2.2, the address space of every process typically includes the entire user space, as well as the entire kernel space, which typically also has all physical memory (in-use) mapped. However, these memory regions are only accessible in privileged mode (cf. Section 2.2).

In this work, we demonstrate leaking secrets by bypassing the privileged-mode isolation, giving an attacker full read access to the entire kernel space, including any physical memory mapped and, thus, the physical memory of any other process and the kernel. Note that Kocher et al. [40] pursue an orthogonal approach, called Spectre Attacks, which trick speculatively executed instructions into leaking information that the victim process is authorized to access. As a result, Spectre Attacks lack the privilege escalation aspect of Meltdown and require tailoring to the victim process's software environment, but apply more broadly to CPUs that support speculative execution and are not prevented by KAISER.

The full Meltdown attack consists of two building blocks, as illustrated

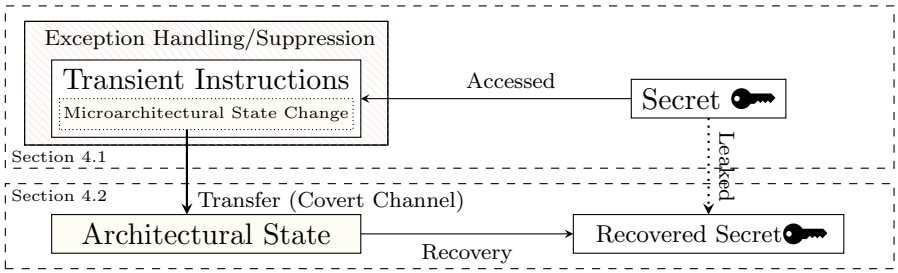


Figure 6.5.: The Meltdown attack uses exception handling or suppression, e.g., TSX, to run a series of transient instructions. These transient instructions obtain a (persistent) secret value and change the microarchitectural state of the processor based on this secret value. This forms the sending part of a microarchitectural covert channel. The receiving side reads the microarchitectural state, making it architectural and recovers the secret value.

in Figure 6.5. The first building block of Meltdown is to make the CPU execute one or more instructions that would never occur in the executed path. In the toy example (cf. Section 3), this is an access to an array, which would normally never be executed, as the previous instruction always raises an exception. We call such an instruction, which is executed out of order and leaving measurable side effects, a *transient instruction*. Furthermore, we call any sequence of instructions containing at least one transient instruction a transient instruction sequence.

In order to leverage transient instructions for an attack, the transient instruction sequence must utilize a secret value that an attacker wants to leak. Section 4.1 describes building blocks to run a transient instruction sequence with a dependency on a secret value.

The second building block of Meltdown is to transfer the microarchitectural side effect of the transient instruction sequence to an architectural state to further process the leaked secret. Thus, the second building described in Section 4.2 describes building blocks to transfer a microarchitectural side effect to an architectural state using a covert channel.

4.1. Executing Transient Instructions

The first building block of Meltdown is the execution of transient instructions. Transient instructions occur all the time, as the CPU continuously runs ahead of the current instruction to minimize the experienced latency and, thus, to maximize the performance (cf. Section 2.1). Transient instructions introduce an exploitable side channel if their operation depends on a secret value. We focus on addresses that are mapped within the attacker's process, *i.e.*, the user-accessible user space addresses as well as the user-inaccessible kernel space addresses. Note that attacks targeting code that is executed within the context (*i.e.*, address space) of another process are possible [40], but out of scope in this work, since all physical memory (including the memory of other processes) can be read through the kernel address space regardless.

Accessing user-inaccessible pages, such as kernel pages, triggers an exception which generally terminates the application. If the attacker targets a secret at a user-inaccessible address, the attacker has to cope with this exception. We propose two approaches: With *exception handling*, we catch the exception effectively occurring after executing the transient instruction sequence, and with *exception suppression*, we prevent the exception from occurring at all and instead redirect the control flow after executing the transient instruction sequence. We discuss these approaches in detail in the following.

Exception handling. A trivial approach is to fork the attacking application before accessing the invalid memory location that terminates the process and only access the invalid memory location in the child process. The CPU executes the transient instruction sequence in the child process before crashing. The parent process can then recover the secret by observing the microarchitectural state, e.g., through a side-channel.

It is also possible to install a signal handler that is executed when a certain exception occurs, e.g., a segmentation fault. This allows the attacker to issue the instruction sequence and prevent the application from crashing, reducing the overhead as no new process has to be created.

Exception suppression. A different approach to deal with exceptions is to prevent them from being raised in the first place. Transactional memory allows to group memory accesses into one seemingly atomic operation,

giving the option to roll-back to a previous state if an error occurs. If an exception occurs within the transaction, the architectural state is reset, and the program execution continues without disruption.

Furthermore, speculative execution issues instructions that might not occur on the executed code path due to a branch misprediction. Such instructions depending on a preceding conditional branch can be speculatively executed. Thus, the invalid memory access is put within a speculative instruction sequence that is only executed if a prior branch condition evaluates to true. By making sure that the condition never evaluates to true in the executed code path, we can suppress the occurring exception as the memory access is only executed speculatively. This technique may require sophisticated training of the branch predictor. Kocher et al. [40] pursue this approach in orthogonal work, since this construct can frequently be found in code of other processes.

4.2. Building a Covert Channel

The second building block of Meltdown is the transfer of the microarchitectural state, which was changed by the transient instruction sequence, into an architectural state (cf. Figure 6.5). The transient instruction sequence can be seen as the sending end of a microarchitectural covert channel. The receiving end of the covert channel receives the microarchitectural state change and deduces the secret from the state. Note that the receiver is not part of the transient instruction sequence and can be a different thread or even a different process e.g., the parent process in the fork-and-crash approach.

We leverage techniques from cache attacks, as the cache state is a microarchitectural state which can be reliably transferred into an architectural state using various techniques [22, 55, 63]. Specifically, we use *Flush+Reload* [63], as it allows to build a fast and low-noise covert channel. Thus, depending on the secret value, the transient instruction sequence (cf. Section 4.1) performs a regular memory access, e.g., as it does in the toy example (cf. Section 3).

After the transient instruction sequence accessed an accessible address, *i.e.*, this is the sender of the covert channel; the address is cached for subsequent accesses. The receiver can then monitor whether the address has been loaded into the cache by measuring the access time to the address. Thus, the sender can transmit a '1'-bit by accessing an address which is

loaded into the monitored cache, and a '0'-bit by not accessing such an address.

Using multiple different cache lines, as in our toy example in Section 3, allows to transmit multiple bits at once. For every of the 256 different byte values, the sender accesses a different cache line. By performing a *Flush+Reload* attack on all of the 256 possible cache lines, the receiver can recover a full byte instead of just one bit. However, since the *Flush+Reload* attack takes much longer (typically several hundred cycles) than the transient instruction sequence, transmitting only a single bit at once is more efficient. The attacker can simply do that by shifting and masking the secret value accordingly.

Note that the covert channel is not limited to microarchitectural states which rely on the cache. Any microarchitectural state which can be influenced by an instruction (sequence) and is observable through a side channel can be used to build the sending end of a covert channel. The sender could, for example, issue an instruction (sequence) which occupies a certain execution port such as the ALU to send a '1'-bit. The receiver measures the latency when executing an instruction (sequence) on the same execution port. A high latency implies that the sender sends a '1'-bit, whereas a low latency implies that sender sends a '0'-bit. The advantage of the *Flush+Reload* cache covert channel is the noise resistance and the high transmission rate [22]. Furthermore, the leakage can be observed from any CPU core [63], *i.e.*, rescheduling events do not significantly affect the covert channel.

5. Meltdown

In this section, we present Meltdown, a powerful attack allowing to read arbitrary physical memory from an unprivileged user program, comprised of the building blocks presented in Section 4. First, we discuss the attack setting to emphasize the wide applicability of this attack. Second, we present an attack overview, showing how Meltdown can be mounted on both Windows and Linux on personal computers, on Android on mobile phones as well as in the cloud. Finally, we discuss a concrete implementation of Meltdown allowing to dump arbitrary kernel memory with 3.2 KB/s to 503 KB/s.

Attack setting. In our attack, we consider personal computers and virtual machines in the cloud. In the attack scenario, the attacker has arbitrary unprivileged code execution on the attacked system, *i.e.*, the attacker can run any code with the privileges of a normal user. However, the attacker has no physical access to the machine. Furthermore, we assume that the system is fully protected with state-of-the-art software-based defenses such as ASLR and KASLR as well as CPU features like SMAP, SMEP, NX, and PXN. Most importantly, we assume a completely bug-free operating system, thus, no software vulnerability exists that can be exploited to gain kernel privileges or leak information. The attacker targets secret user data, e.g., passwords and private keys, or any other valuable information.

5.1. Attack Description

Meltdown combines the two building blocks discussed in Section 4. First, an attacker makes the CPU execute a transient instruction sequence which uses an inaccessible secret value stored somewhere in physical memory (cf. Section 4.1). The transient instruction sequence acts as the transmitter of a covert channel (cf. Section 4.2), ultimately leaking the secret value to the attacker.

Meltdown consists of 3 steps:

Step 1 The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.

Step 2 A transient instruction accesses a cache line based on the secret content of the register.

Step 3 The attacker uses *Flush+Reload* to determine the accessed cache line and hence the secret stored at the chosen memory location.

By repeating these steps for different memory locations, the attacker can dump the kernel memory, including the entire physical memory.

Listing 6.2 shows the basic implementation of the transient instruction sequence and the sending part of the covert channel, using x86 assembly instructions. Note that this part of the attack could also be implemented entirely in higher level languages like C. In the following, we will discuss each step of Meltdown and the corresponding code line in Listing 6.2.

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Listing 6.2: The core of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. Subsequent instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access.

Step 1: Reading the secret. To load data from the main memory into a register, the data in the main memory is referenced using a virtual address. In parallel to translating a virtual address into a physical address, the CPU also checks the permission bits of the virtual address, *i.e.*, whether this virtual address is user accessible or only accessible by the kernel. As already discussed in Section 2.2, this hardware-based isolation through a permission bit is considered secure and recommended by the hardware vendors. Hence, modern operating systems always map the entire kernel into the virtual address space of every user process.

As a consequence, all kernel addresses lead to a valid physical address when translating them, and the CPU can access the content of such addresses. The only difference to accessing a user space address is that the CPU raises an exception as the current permission level does not allow to access such an address. Hence, the user space cannot simply read the contents of such an address. However, Meltdown exploits the out-of-order execution of modern CPUs, which still executes instructions in the small time window between the illegal memory access and the raising of the exception.

In line 4 of Listing 6.2, we load the byte value located at the target kernel address, stored in the `RCX` register, into the least significant byte of the `RAX` register represented by `AL`. As explained in more detail in Section 2.1, the `MOV` instruction is fetched by the core, decoded into μOPs, allocated, and sent to the reorder buffer. There, architectural registers (e.g., `RAX` and `RCX` in Listing 6.2) are mapped to underlying physical registers enabling out-of-order execution. Trying to utilize the pipeline as much as possible, subsequent instructions (lines 5-7) are already decoded and allocated

as μOPs as well. The μOPs are further sent to the reservation station holding the μOPs while they wait to be executed by the corresponding execution unit. The execution of a μOP can be delayed if execution units are already used to their corresponding capacity, or operand values have not been computed yet.

When the kernel address is loaded in line 4, it is likely that the CPU already issued the subsequent instructions as part of the out-of-order execution, and that their corresponding μOPs wait in the reservation station for the content of the kernel address to arrive. As soon as the fetched data is observed on the common data bus, the μOPs can begin their execution. Furthermore, processor interconnects [3, 31] and cache coherence protocols [59] guarantee that the most recent value of a memory address is read, regardless of the storage location in a multi-core or multi-CPU system.

When the μOPs finish their execution, they retire in-order, and, thus, their results are committed to the architectural state. During the retirement, any interrupts and exceptions that occurred during the execution of the instruction are handled. Thus, if the `MOV` instruction that loads the kernel address is retired, the exception is registered, and the pipeline is flushed to eliminate all results of subsequent instructions which were executed out of order. However, there is a race condition between raising this exception and our attack step 2 as described below.

As reported by Gruss et al. [21], prefetching kernel addresses sometimes succeeds. We found that prefetching the kernel address can slightly improve the performance of the attack on some systems.

Step 2: Transmitting the secret. The instruction sequence from step 1 which is executed out of order has to be chosen in a way that it becomes a transient instruction sequence. If this transient instruction sequence is executed before the `MOV` instruction is retired (*i.e.*, raises the exception), and the transient instruction sequence performed computations based on the secret, it can be utilized to transmit the secret to the attacker.

As already discussed, we utilize cache attacks that allow building fast and low-noise covert channels using the CPU's cache. Thus, the transient instruction sequence has to encode the secret into the microarchitectural cache state, similar to the toy example in Section 3.

We allocate a probe array in memory and ensure that no part of this

array is cached. To transmit the secret, the transient instruction sequence contains an indirect memory access to an address which is computed based on the secret (inaccessible) value. In line 5 of Listing 6.2, the secret value from step 1 is multiplied by the page size, *i.e.*, 4 KB. The multiplication of the secret ensures that accesses to the array have a large spatial distance to each other. This prevents the hardware prefetcher from loading adjacent memory locations into the cache as well. Here, we read a single byte at once. Hence, our probe array is 256×4096 bytes, assuming 4 KB pages.

Note that in the out-of-order execution we have a noise-bias towards register value '0'. We discuss the reasons for this in Section 5.2. However, for this reason, we introduce a retry-logic into the transient instruction sequence. In case we read a '0', we try to reread the secret (step 1). In line 7, the multiplied secret is added to the base address of the probe array, forming the target address of the covert channel. This address is read to cache the corresponding cache line. The address will be loaded into the L1 data cache of the requesting core and, due to the inclusiveness, also the L3 cache where it can be read from other cores. Consequently, our transient instruction sequence affects the cache state based on the secret value that was read in step 1.

Since the transient instruction sequence in step 2 races against raising the exception, reducing the runtime of step 2 can significantly improve the performance of the attack. For instance, taking care that the address translation for the probe array is cached in the translation-lookaside buffer (TLB) increases the attack performance on some systems.

Step 3: Receiving the secret. In step 3, the attacker recovers the secret value (step 1) by leveraging a microarchitectural side-channel attack (*i.e.*, the receiving end of a microarchitectural covert channel) that transfers the cache state (step 2) back into an architectural state. As discussed in Section 4.2, our implementation of Meltdown relies on *Flush+Reload* for this purpose.

When the transient instruction sequence of step 2 is executed, exactly one cache line of the probe array is cached. The position of the cached cache line within the probe array depends only on the secret which is read in step 1. Thus, the attacker iterates over all 256 pages of the probe array and measures the access time for every first cache line (*i.e.*, offset) on the page. The number of the page containing the cached cache line

corresponds directly to the secret value.

Dumping the entire physical memory. Repeating all 3 steps of Meltdown, an attacker can dump the entire memory by iterating over all addresses. However, as the memory access to the kernel address raises an exception that terminates the program, we use one of the methods from Section 4.1 to handle or suppress the exception.

As all major operating systems also typically map the entire physical memory into the kernel address space (cf. Section 2.2) in every user process, Meltdown can also read the entire physical memory of the target machine.

5.2. Optimizations and Limitations

Inherent bias towards 0. While CPUs generally stall if a value is not available during an out-of-order load operation [28], CPUs might continue with the out-of-order execution by assuming a value for the load [12]. We observed that the illegal memory load in our Meltdown implementation (line 4 in Listing 6.2) often returns a '0', which can be clearly observed when implemented using an `add` instruction instead of the `mov`. The reason for this bias to '0' may either be that the memory load is masked out by a failed permission check, or a speculated value because the data of the stalled load is not available yet.

This inherent bias results from the race condition in the out-of-order execution, which may be won (*i.e.*, reads the correct value), but is often lost (*i.e.*, reads a value of '0'). This bias varies between different machines as well as hardware and software configurations and the specific implementation of Meltdown. In an unoptimized version, the probability that a value of '0' is erroneously returned is high. Consequently, our Meltdown implementation performs a certain number of retries when the code in Listing 6.2 results in reading a value of '0' from the *Flush+Reload* attack. The maximum number of retries is an optimization parameter influencing the attack performance and the error rate. On the Intel Core i5-6200U using exeception handling, we read a '0' on average in 5.25 % ($\sigma = 4.15$) with our unoptimized version. With a simple retry loop, we reduced the probability to 0.67 % ($\sigma = 1.47$). On the Core i7-8700K, we read on average a '0' in 1.78 % ($\sigma = 3.07$). Using Intel TSX, the probability is further reduced to 0.008 %.

Optimizing the case of 0. Due to the inherent bias of Meltdown, a cache hit on cache line '0' in the *Flush+Reload* measurement, does not provide the attacker with any information. Hence, measuring cache line '0' can be omitted and in case there is no cache hit on any other cache line, the value can be assumed to be '0'. To minimize the number of cases where no cache hit on a non-zero line occurs, we retry reading the address in the transient instruction sequence until it encounters a value different from '0' (line 6). This loop is terminated either by reading a non-zero value or by the raised exception of the invalid memory access. In either case, the time until exception handling or exception suppression returns the control flow is independent of the loop after the invalid memory access, *i.e.*, the loop does not slow down the attack measurably. Hence, these optimizations may increase the attack performance.

Single-bit transmission. In the attack description in Section 5.1, the attacker transmitted 8 bits through the covert channel at once and performed $2^8 = 256$ *Flush+Reload* measurements to recover the secret. However, there is a trade-off between running more transient instruction sequences and performing more *Flush+Reload* measurements. The attacker could transmit an arbitrary number of bits in a single transmission through the covert channel, by reading more bits using a `MOV` instruction for a larger data value. Furthermore, the attacker could mask bits using additional instructions in the transient instruction sequence. We found the number of additional instructions in the transient instruction sequence to have a negligible influence on the performance of the attack.

The performance bottleneck in the generic attack described above is indeed, the time spent on *Flush+Reload* measurements. In fact, with this implementation, almost the entire time is spent on *Flush+Reload* measurements. By transmitting only a single bit, we can omit all but one *Flush+Reload* measurement, *i.e.*, the measurement on cache line 1. If the transmitted bit was a '1', then we observe a cache hit on cache line 1. Otherwise, we observe no cache hit on cache line 1.

Transmitting only a single bit at once also has drawbacks. As described above, our side channel has a bias towards a secret value of '0'. If we read and transmit multiple bits at once, the likelihood that all bits are '0' may be quite small for actual user data. The likelihood that a single bit is '0' is typically close to 50 %. Hence, the number of bits read and transmitted at once is a trade-off between some implicit error-reduction and the overall transmission rate of the covert channel.

However, since the error rates are quite small in either case, our evaluation (cf. Section 6) is based on the single-bit transmission mechanics.

Exception Suppression using Intel TSX. In Section 4.1, we discussed the option to prevent that an exception is raised due an invalid memory access. Using Intel TSX, a hardware transactional memory implementation, we can completely suppress the exception [37].

With Intel TSX, multiple instructions can be grouped to a transaction, which appears to be an atomic operation, *i.e.*, either all or no instruction is executed. If one instruction within the transaction fails, already executed instructions are reverted, but no exception is raised.

If we wrap the code from listing 6.2 with such a TSX instruction, any exception is suppressed. However, the microarchitectural effects are still visible, *i.e.*, the cache state is persistently manipulated from within the hardware transaction [19]. This results in higher channel capacity, as suppressing the exception is significantly faster than trapping into the kernel for handling the exception, and continuing afterward.

Dealing with KASLR. In 2013, kernel address space layout randomization (KASLR) was introduced to the Linux kernel (starting from version 3.14 [11]) allowing to randomize the location of kernel code at boot time. However, only as recently as May 2017, KASLR was enabled by default in version 4.12 [54]. With KASLR also the direct-physical map is randomized and not fixed at a certain address such that the attacker is required to obtain the randomized offset before mounting the Meltdown attack. However, the randomization is limited to 40 bit.

Thus, if we assume a setup of the target machine with 8 GB of RAM, it is sufficient to test the address space for addresses in 8 GB steps. This allows covering the search space of 40 bit with only 128 tests in the worst case. If the attacker can successfully obtain a value from a tested address, the attacker can proceed to dump the entire memory from that location. This allows mounting Meltdown on a system despite being protected by KASLR within seconds.

Table 6.1.: Experimental setups.

Environment	CPU Model	Cores
Lab	Celeron G540	2
Lab	Core i5-3230M	2
Lab	Core i5-3320M	2
Lab	Core i7-4790	4
Lab	Core i5-6200U	2
Lab	Core i7-6600U	2
Lab	Core i7-6700K	4
Lab	Core i7-8700K	12
Lab	Xeon E5-1630 v3	8
Cloud	Xeon E5-2676 v3	12
Cloud	Xeon E5-2650 v4	12
Phone	Exynos 8890	8

6. Evaluation

In this section, we evaluate Meltdown and the performance of our proof-of-concept implementation.[11] Section 6.1 discusses the information which Meltdown can leak, and Section 6.2 evaluates the performance of Meltdown, including countermeasures. Finally, we discuss limitations for AMD and ARM in Section 6.3.

Table 6.1 shows a list of configurations on which we successfully reproduced Meltdown. For the evaluation of Meltdown, we used both laptops as well as desktop PCs with Intel Core CPUs and an ARM-based mobile phone. For the cloud setup, we tested Meltdown in virtual machines running on Intel Xeon CPUs hosted in the Amazon Elastic Compute Cloud as well as on DigitalOcean. Note that for ethical reasons we did not use Meltdown on addresses referring to physical memory of other tenants.

6.1. Leakage and Environments

We evaluated Meltdown on both Linux (cf. Section 6.1.1), Windows 10 (cf. Section 6.1.3) and Android (cf. Section 6.1.4), without the patches introducing the KAISER mechanism. On these operating systems, Meltdown can successfully leak kernel memory. We also evaluated the effect of the KAISER patches on Meltdown on Linux, to show that KAISER prevents the leakage of kernel memory (cf. Section 6.1.2). Furthermore,

[11] `https://github.com/IAIK/meltdown`

we discuss the information leakage when running inside containers such as Docker (cf. Section 6.1.5). Finally, we evaluate Meltdown on uncached and uncacheable memory (cf. Section 6.1.6).

6.1.1. Linux

We successfully evaluated Meltdown on multiple versions of the Linux kernel, from 2.6.32 to 4.13.0, without the patches introducing the KAISER mechanism. On all these versions of the Linux kernel, the kernel address space is also mapped into the user address space. Thus, all kernel addresses are also mapped into the address space of user space applications, but any access is prevented due to the permission settings for these addresses. As Meltdown bypasses these permission settings, an attacker can leak the complete kernel memory if the virtual address of the kernel base is known. Since all major operating systems also map the entire physical memory into the kernel address space (cf. Section 2.2), all physical memory can also be read.

Before kernel 4.12, kernel address space layout randomization (KASLR) was not active by default [57]. If KASLR is active, Meltdown can still be used to find the kernel by searching through the address space (cf. Section 5.2). An attacker can also simply de-randomize the direct-physical map by iterating through the virtual address space. Without KASLR, the direct-physical map starts at address `0xffff 8800 0000 0000` and linearly maps the entire physical memory. On such systems, an attacker can use Meltdown to dump the entire physical memory, simply by reading from virtual addresses starting at `0xffff 8800 0000 0000`.

On newer systems, where KASLR is active by default, the randomization of the direct-physical map is limited to 40 bit. It is even further limited due to the linearity of the mapping. Assuming that the target system has at least 8 GB of physical memory, the attacker can test addresses in steps of 8 GB, resulting in a maximum of 128 memory locations to test. Starting from one discovered location, the attacker can again dump the entire physical memory.

Hence, for the evaluation, we can assume that the randomization is either disabled, or the offset was already retrieved in a pre-computation step.

6.1.2. Linux with KAISER Patch

The KAISER patch by Gruss et al. [20] implements a stronger isolation between kernel and user space. KAISER does not map any kernel memory in the user space, except for some parts required by the x86 architecture (e.g., interrupt handlers). Thus, there is no valid mapping to either kernel memory or physical memory (via the direct-physical map) in the user space, and such addresses can therefore not be resolved. Consequently, Meltdown cannot leak any kernel or physical memory except for the few memory locations which have to be mapped in user space.

We verified that KAISER indeed prevents Meltdown, and there is no leakage of any kernel or physical memory.

Furthermore, if KASLR is active, and the few remaining memory locations are randomized, finding these memory locations is not trivial due to their small size of several kilobytes. Section 7.2 discusses the security implications of these mapped memory locations.

6.1.3. Microsoft Windows

We successfully evaluated Meltdown on a recent Microsoft Windows 10 operating system, last updated just before patches against Meltdown were rolled out. In line with the results on Linux (cf. Section 6.1.1), Meltdown also can leak arbitrary kernel memory on Windows. This is not surprising, since Meltdown does not exploit any software issues, but is caused by a hardware issue.

In contrast to Linux, Windows does not have the concept of an identity mapping, which linearly maps the physical memory into the virtual address space. Instead, a large fraction of the physical memory is mapped in the paged pools, non-paged pools, and the system cache. Furthermore, Windows maps the kernel into the address space of every application too. Thus, Meltdown can read kernel memory which is mapped in the kernel address space, *i.e.*, any part of the kernel which is not swapped out, and any page mapped in the paged and non-paged pool, and the system cache.

Note that there are physical pages which are mapped in one process but not in the (kernel) address space of another process, *i.e.*, physical pages which cannot be attacked using Meltdown. However, most of the physical memory will still be accessible through Meltdown.

We were successfully able to read the binary of the Windows kernel using

Meltdown. To verify that the leaked data is actual kernel memory, we first used the Windows kernel debugger to obtain kernel addresses containing actual data. After leaking the data, we again used the Windows kernel debugger to compare the leaked data with the actual memory content, confirming that Meltdown can successfully leak kernel memory.

6.1.4. Android

We successfully evaluated Meltdown on a Samsung Galaxy S7 mohile phone running LineageOS Android 14.1 with a Linux kernel 3.18.14. The device is equipped with a Samsung Exynos 8 Octa 8890 SoC consisting of a ARM Cortex-A53 CPU with 4 cores as well as an Exynos M1 "Mongoose" CPU with 4 cores [6]. While we were not able to mount the attack on the Cortex-A53 CPU, we successfully mounted Meltdown on Samsung's custom cores. Using *exception suppression* described in Section 4.1, we successfully leaked a pre-defined string using the direct-physical map located at the virtual address `0xffff ffbf c000 0000`.

6.1.5. Containers

We evaluated Meltdown in containers sharing a kernel, including Docker, LXC, and OpenVZ and found that the attack can be mounted without any restrictions. Running Meltdown inside a container allows to leak information not only from the underlying kernel but also from all other containers running on the same physical host.

The commonality of most container solutions is that every container uses the same kernel, *i.e.*, the kernel is shared among all containers. Thus, every container has a valid mapping of the entire physical memory through the direct-physical map of the shared kernel. Furthermore, Meltdown cannot be blocked in containers, as it uses only memory accesses. Especially with Intel TSX, only unprivileged instructions are executed without even trapping into the kernel.

Thus, the isolation of containers sharing a kernel can be entirely broken using Meltdown. This is especially critical for cheaper hosting providers where users are not separated through fully virtualized machines, but only through containers. We verified that our attack works in such a setup, by successfully leaking memory contents from a container of a different user under our control.

6.1.6. Uncached and Uncacheable Memory

In this section, we evaluate whether it is a requirement for data to be leaked by Meltdown to reside in the L1 data cache [32]. Therefore, we constructed a setup with two processes pinned to different physical cores. By flushing the value, using the `clflush` instruction, and only reloading it on the other core, we create a situation where the target data is not in the L1 data cache of the attacker core. As described in Section 6.2, we can still leak the data at a lower reading rate. This clearly shows that data presence in the attacker's L1 data cache is not a requirement for Meltdown. Furthermore, this observation has also been confirmed by other researchers [5, 7, 35].

The reason why Meltdown can leak uncached memory may be that Meltdown implicitly caches the data. We devise a second experiment, where we mark pages as *uncacheable* and try to leak data from them. This has the consequence that every read or write operation to one of those pages will directly go to the main memory, thus, bypassing the cache. In practice, only a negligible amount of system memory is marked uncacheable. We observed that if the attacker is able to trigger a legitimate load of the target address, e.g., by issuing a system call (regular or in speculative execution [40]), on the same CPU core as the Meltdown attack, the attacker can leak the content of the uncacheable pages. We suspect that Meltdown reads the value from the line fill buffers. As the fill buffers are shared between threads running on the same core, the read to the same address within the Meltdown attack could be served from one of the fill buffers allowing the attack to succeed. However, we leave further investigations on this matter open for future work.

A similar observation on uncacheable memory was also made with Spectre attacks on the System Management Mode [10]. While the attack works on memory set uncacheable over Memory-Type Range Registers, it does not work on memory-mapped I/O regions, which is the expected behavior as accesses to memory-mapped I/O can always have architectural effects.

6.2. Meltdown Performance

To evaluate the performance of Meltdown, we leaked known values from kernel memory. This allows us to not only determine how fast an attacker can leak memory, but also the error rate, *i.e.*, how many byte errors to expect. The race condition in Meltdown (cf. Section 5.2) has a significant

influence on the performance of the attack, however, the race condition can always be won. If the targeted data resides close to the core, e.g., in the L1 data cache, the race condition is won with a high probability. In this scenario, we achieved average reading rates of up to 582 KB/s ($\mu = 552.4, \sigma = 10.2$) with an error rate as low as 0.003 % ($\mu = 0.009, \sigma = 0.014$) using exception suppression on the Core i7-8700K over 10 runs over 10 seconds. With the Core i7-6700K we achieved 569 KB/s ($\mu = 515.5, \sigma = 5.99$) with an minimum error rate of 0.002 % ($\mu = 0.003, \sigma = 0.001$) and 491 KB/s ($\mu = 466.3, \sigma = 16.75$) with a minimum error rate of 10.7 % ($\mu = 11.59, \sigma = 0.62$) on the Xeon E5-1630. However, with a slower version with an average reading speed of 137 KB/s, we were able to reduce the error rate to 0. Furthermore, on the Intel Core i7-6700K if the data resides in the L3 data cache but not in L1, the race condition can still be won often, but the average reading rate decreases to 12.4 KB/s with an error rate as low as 0.02 % using exception suppression. However, if the data is uncached, winning the race condition is more difficult and, thus, we have observed reading rates of less than 10 B/s on most systems. Nevertheless, there are two optimizations to improve the reading rate: First, by simultaneously letting other threads prefetch the memory locations [21] of and around the target value and access the target memory location (with exception suppression or handling). This increases the probability that the spying thread sees the secret data value in the right moment during the data race. Second, by triggering the hardware prefetcher through speculative accesses to memory locations of and around the target value. With these two optimizations, we can improve the reading rate for uncached data to 3.2 KB/s.

For all tests, we used *Flush+Reload* as a covert channel to leak the memory as described in Section 5, and Intel TSX to suppress the exception. An extensive evaluation of exception suppression using conditional branches was done by Kocher et al. [40] and is thus omitted in this paper for the sake of brevity.

6.3. Limitations on ARM and AMD

We also tried to reproduce the Meltdown bug on several ARM and AMD CPUs. While we were able to successfully leak kernel memory with the attack described in Section 5 on different Intel CPUs and a Samsung Exynos M1 processor, we did not manage to mount Meltdown on other ARM cores nor on AMD. In the case of ARM, the only affected proces-

sor is the Cortex-A75 [17] which has not been available and, thus, was not among our devices under test. However, appropriate kernel patches have already been provided [2]. Furthermore, an altered attack of Meltdown targeting system registers instead of inaccessible memory locations is applicable on several ARM processors [17]. Meanwhile, AMD publicly stated that none of their CPUs are not affected by Meltdown due to architectural differences [1].

The major part of a microarchitecture is usually not publicly documented. Thus, it is virtually impossible to know the differences in the implementations that allow or prevent Meltdown without proprietary knowledge and, thus, the intellectual property of the individual CPU manufacturers. The key point is that on a microarchitectural level the load to the unprivileged address and the subsequent instructions are executed while the fault is only handled when the faulting instruction is retired. It can be assumed that the execution units for the load and the TLB are designed differently on ARM, AMD and Intel and, thus, the privileges for the load are checked differently and occurring faults are handled differently, e.g., issuing a load only after the permission bit in the page table entry has been checked. However, from a performance perspective, issuing the load in parallel or only checking permissions while retiring an instruction is a reasonable decision. As trying to load kernel addresses from user space is not what programs usually do and by guaranteeing that the state does not become architecturally visible, not squashing the load is legitimate. However, as the state becomes visible on the microarchitectural level, such implementations are vulnerable.

However, for both ARM and AMD, the toy example as described in Section 3 works reliably, indicating that out-of-order execution generally occurs and instructions past illegal memory accesses are also performed.

7. Countermeasures

In this section, we discuss countermeasures against the Meltdown attack. At first, as the issue is rooted in the hardware itself, we discuss possible microcode updates and general changes in the hardware design. Second, we discuss the KAISER countermeasure that has been developed to mitigate side-channel attacks against KASLR which inadvertently also protects against Meltdown.

7.1. Hardware

Meltdown bypasses the hardware-enforced isolation of security domains. There is no software vulnerability involved in Meltdown. Any software patch (e.g., KAISER [20]) will leave small amounts of memory exposed (cf. Section 7.2). There is no documentation whether a fix requires the development of completely new hardware, or can be fixed using a microcode update.

As Meltdown exploits out-of-order execution, a trivial countermeasure is to disable out-of-order execution completely. However, performance impacts would be devastating, as the parallelism of modern CPUs could not be leveraged anymore. Thus, this is not a viable solution.

Meltdown is some form of race condition between the fetch of a memory address and the corresponding permission check for this address. Serializing the permission check and the register fetch can prevent Meltdown, as the memory address is never fetched if the permission check fails. However, this involves a significant overhead to every memory fetch, as the memory fetch has to stall until the permission check is completed.

A more realistic solution would be to introduce a hard split of user space and kernel space. This could be enabled optionally by modern kernels using a new hard-split bit in a CPU control register, e.g., CR4. If the hard-split bit is set, the kernel has to reside in the upper half of the address space, and the user space has to reside in the lower half of the address space. With this hard split, a memory fetch can immediately identify whether such a fetch of the destination would violate a security boundary, as the privilege level can be directly derived from the virtual address without any further lookups. We expect the performance impacts of such a solution to be minimal. Furthermore, the backwards compatibility is ensured, since the hard-split bit is not set by default and the kernel only sets it if it supports the hard-split feature.

Note that these countermeasures only prevent Meltdown, and not the class of Spectre attacks described by Kocher et al. [40]. Likewise, their presented countermeasures [40] do not affect Meltdown. We stress that it is important to deploy countermeasures against both attacks.

7.2. KAISER

As existing hardware is not as easy to patch, there is a need for software workarounds until new hardware can be deployed. Gruss et al. [20] proposed KAISER, a kernel modification to not have the kernel mapped in the user space. This modification was intended to prevent side-channel attacks breaking KASLR [21, 29, 37]. However, it also prevents Meltdown, as it ensures that there is no valid mapping to kernel space or physical memory available in user space. In concurrent work to KAISER, Gens et al. [14] proposed LAZARUS as a modification to the Linux kernel to thwart side-channel attacks breaking KASLR by separating address spaces similar to KAISER. As the Linux kernel continued the development of the original KAISER patch and Windows [53] and macOS [34] based their implementation on the concept of KAISER to defeat Meltdown, we will discuss KAISER in more depth.

Although KAISER provides basic protection against Meltdown, it still has some limitations. Due to the design of the x86 architecture, several privileged memory locations are still required to be mapped in user space [20], leaving a residual attack surface for Meltdown, *i.e.*, these memory locations can still be read from user space. Even though these memory locations do not contain any secrets, e.g., credentials, they might still contain pointers. Leaking one pointer can suffice to break KASLR, as the randomization can be computed from the pointer value.

Still, KAISER is the best short-time solution currently available and should therefore be deployed on all systems immediately. Even with Meltdown, KAISER can avoid having any kernel pointers on memory locations that are mapped in the user space which would leak information about the randomized offsets. This would require trampoline locations for every kernel pointer, *i.e.*, the interrupt handler would not call into kernel code directly, but through a trampoline function. The trampoline function must only be mapped in the kernel. It must be randomized with a different offset than the remaining kernel. Consequently, an attacker can only leak pointers to the trampoline code, but not the randomized offsets of the remaining kernel. Such trampoline code is required for every kernel memory that still has to be mapped in user space and contains kernel addresses. This approach is a trade-off between performance and security which has to be assessed in future work.

The original KAISER patch [18] for the Linux kernel has been improved [24–27] with various optimizations, e.g., support for PCIDs. Afterwards, be-

fore merging it into the mainline kernel, it has been renamed to kernel page-table isolation (KPTI) [15, 49]. KPTI is active in recent releases of the Linux kernel and has been backported to older versions as well [30, 42–44].

Microsoft implemented a similar patch inspired by KAISER [53] named KVA Shadow [39]. While KVA Shadow only maps a minimum of kernel transition code and data pages required to switch between address spaces, it does not protect against side-channel attacks against KASLR [39].

Apple released updates in iOS 11.2, macOS 10.13.2 and tvOS 11.2 to mitigate Meltdown. Similar to Linux and Windows, macOS shared the kernel and user address spaces in 64-bit mode unless the `-no-shared-cr3` boot option was set [46]. This option unmaps the user space while running in kernel mode but does not unmap the kernel while running in user mode [51]. Hence, it has no effect on Meltdown. Consequently, Apple introduced *Double Map* [34] following the principles of KAISER to mitigate Meltdown.

8. Discussion

Meltdown fundamentally changes our perspective on the security of hardware optimizations that manipulate the state of microarchitectural elements. The fact that hardware optimizations can change the state of microarchitectural elements, and thereby imperil secure software implementations, is known since more than 20 years [41]. Both industry and the scientific community so far accepted this as a necessary evil for efficient computing. Today it is considered a bug when a cryptographic algorithm is not protected against the microarchitectural leakage introduced by the hardware optimizations. Meltdown changes the situation entirely. Meltdown shifts the granularity from a comparably low spatial and temporal granularity, e.g., 64-bytes every few hundred cycles for cache attacks, to an arbitrary granularity, allowing an attacker to read every single bit. This is nothing any (cryptographic) algorithm can protect itself against. KAISER is a short-term software fix, but the problem we have uncovered is much more significant.

We expect several more performance optimizations in modern CPUs which affect the microarchitectural state in some way, not even necessarily through the cache. Thus, hardware which is designed to provide certain security guarantees, e.g., CPUs running untrusted code, requires a

redesign to avoid Meltdown- and Spectre-like attacks. Meltdown also shows that even error-free software, which is explicitly written to thwart side-channel attacks, is not secure if the design of the underlying hardware is not taken into account.

With the integration of KAISER into all major operating systems, an important step has already been done to prevent Meltdown. KAISER is a fundamental change in operating system design. Instead of always mapping everything into the address space, mapping only the minimally required memory locations appears to be a first step in reducing the attack surface. However, it might not be enough, and even stronger isolation may be required. In this case, we can trade flexibility for performance and security, by e.g., enforcing a certain virtual memory layout for every operating system. As most modern operating systems already use a similar memory layout, this might be a promising approach.

Meltdown also heavily affects cloud providers, especially if the guests are not fully virtualized. For performance reasons, many hosting or cloud providers do not have an abstraction layer for virtual memory. In such environments, which typically use containers, such as Docker or OpenVZ, the kernel is shared among all guests. Thus, the isolation between guests can simply be circumvented with Meltdown, fully exposing the data of all other guests on the same host. For these providers, changing their infrastructure to full virtualization or using software workarounds such as KAISER would both increase the costs significantly.

Concurrent work has investigated the possibility to read kernel memory via out-of-order or speculative execution, but has not succeeded [13, 50]. We are the first to demonstrate that it is possible. Even if Meltdown is fixed, Spectre [40] will remain an issue, requiring different defenses. Mitigating only one of them will leave the security of the entire system at risk. Meltdown and Spectre open a new field of research to investigate to what extent performance optimizations change the microarchitectural state, how this state can be translated into an architectural state, and how such attacks can be prevented.

9. Conclusion

In this paper, we presented Meltdown, a novel software-based attack exploiting out-of-order execution and side channels on modern processors to read arbitrary kernel memory from an unprivileged user space program.

Without requiring any software vulnerability and independent of the operating system, Meltdown enables an adversary to read sensitive data of other processes or virtual machines in the cloud with up to 503 KB/s, affecting millions of devices. We showed that the countermeasure KAISER, originally proposed to protect from side-channel attacks against KASLR, inadvertently impedes Meltdown as well. We stress that KAISER needs to be deployed on every operating system as a short-term workaround, until Meltdown is fixed in hardware, to prevent large-scale exploitation of Meltdown.

Acknowledgments

Several authors of this paper found Meltdown independently, ultimately leading to this collaboration. We want to thank everyone who helped us in making this collaboration possible, especially Intel who handled our responsible disclosure professionally, comunicated a clear timeline and connected all involved researchers. We thank Mark Brand from Google Project Zero for contributing ideas and Peter Cordes and Henry Wong for valuable feedback. We would like to thank our anonymous reviewers for their valuable feedback. Furthermore, we would like to thank Intel, ARM, Qualcomm, and Microsoft for feedback on an early draft.

Daniel Gruss, Moritz Lipp, Stefan Mangard and Michael Schwarz were supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

Daniel Genkin was supported by NSF awards #1514261 and #1652259, financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017-2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

References

[1] AMD. *Software techniques for managing speculation on AMD processors*. 2018.

[2] ARM. *AArch64 Linux kernel port (KPTI base)*. 2018. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/arm64/linux.git/log/?h=kpti`.

[3] ARM Limited. *ARM CoreLink CCI-400 Cache Coherent Interconnect Technical Reference Manual*. r1p5. ARM Limited, 2015.

[4] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. "'Ooh Aah... Just a Little Bit': A small amount of side channel can go a long way". In: *CHES'14*. 2014.

[5] Pavel Boldin. *Meltdown Reading Other process's memory*. Jan. 2018. URL: https://www.youtube.com/watch?v=EMBGXswJC4s.

[6] Brad Burgess. "Samsung Exynos M1 Processor". In: *IEEE Hot Chips*. 2016. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7936205.

[7] Raphael Carvalho. *Twitter: Meltdown with Uncached Memory*. Jan. 2018. URL: https://twitter.com/raphael_scarv/status/952078140028964864.

[8] Chih-Cheng Cheng. "The schemes and performances of dynamic branch predictors". In: *Berkeley Wireless Research Center, Tech. Rep* (2000).

[9] Advanced Micro Devies. *AMD Takes Computing to a New Horizon with Ryzen™Processors*. 2016. URL: https://www.amd.com/en-us/press-releases/Pages/amd-takes-computing-2016dec13.aspx.

[10] Eclypsium. *System Management Mode Speculative Execution Attacks*. May 2018. URL: https://blog.eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/.

[11] Jake Edge. *Kernel address space layout randomization*. 2013. URL: https://lwn.net/Articles/569635/.

[12] R. Eickemeyer, H. Le, D. Nguyen, B. Stolt, and B. Thompto. *Load lookahead prefetch for microprocessors*. US Patent App. 11/016,236. 2006. URL: https://encrypted.google.com/patents/US20060149935.

[13] Anders Fogh. *Negative Result: Reading Kernel Memory From User Mode*. 2017. URL: https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/.

[14] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. "LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization". In: *RAID*. 2017.

[15] Thomas Gleixner. *x86/kpti: Kernel Page Table Isolation (was KAISER)*. Dec. 2017. URL: https://lkml.org/lkml/2017/12/4/709.

[16] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. “ASLR on the Line: Practical Cache Attacks on the MMU”. In: *NDSS*. 2017.

[17] Richard Grisenthwaite. *Cache Speculation Side-channels*. 2018.

[18] Daniel Gruss. *[RFC, PATCH] x86_64: KAISER - do not map kernel in user mode*. May 2017. URL: https://lkml.org/lkml/2017/5/4/220.

[19] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. “Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory”. In: *USENIX Security Symposium*. 2017.

[20] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. “KASLR is Dead: Long Live KASLR”. In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2017, pp. 161–176.

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *CCS*. 2016.

[22] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA*. 2016.

[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

[24] Dave Hansen. *[PATCH 00/23] KAISER: unmap most of the kernel from userspace page tables*. Oct. 2017. URL: https://lkml.org/lkml/2017/10/31/884.

[25] Dave Hansen. *[v2] KAISER: unmap most of the kernel from userspace page tables*. Nov. 2017. URL: https://lkml.org/lkml/2017/11/8/752.

[26] Dave Hansen. *[v3] KAISER: unmap most of the kernel from userspace page tables*. Nov. 2017. URL: https://lkml.org/lkml/2017/11/10/433.

[27] Dave Hansen. *[v4] KAISER: unmap most of the kernel from userspace page tables*. Nov. 2017. URL: https://lkml.org/lkml/2017/11/22/956.

[28] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufmann, 2017.

[29] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR". In: *S&P*. 2013.

[30] Ben Hutchings. *Linux 3.16.53*. 2018. URL: https://cdn.kernel.org/pub/linux/kernel/v3.x/ChangeLog-3.16.53.

[31] Intel. *An Introduction to the Intel QuickPath Interconnect*. Jan. 2009.

[32] Intel. *Intel Analysis of Speculative Execution Side Channels*. Jan. 2018. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[33] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2017.

[34] Alex Ionescu. *Twitter: Apple Double Map*. 2017. URL: https://twitter.com/aionescu/status/948609809540046849.

[35] Alex Ionescu. *Twitter: Meltdown with Uncached Memory*. Jan. 2018. URL: https://twitter.com/aionescu/status/950994906759143425.

[36] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a minute! A fast, Cross-VM attack on AES". In: *RAID'14*. 2014.

[37] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX". In: *CCS*. 2016.

[38] Daniel A Jiménez and Calvin Lin. "Dynamic branch prediction with perceptrons". In: *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE. 2001, pp. 197–206.

[39] Ken Johnson. *KVA Shadow: Mitigating Meltdown on Windows*. Mar. 2018. URL: https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/.

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P*. 2019.

[41] Paul C. Kocher. "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems". In: *CRYPTO*. 1996.

[42] Greg Kroah-Hartman. *Linux 4.14.11*. 2018. URL: https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.14.11.

[43] Greg Kroah-Hartman. *Linux 4.4.110*. 2018. URL: https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.4.110.

[44] Greg Kroah-Hartman. *Linux 4.9.75*. 2018. URL: https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.9.75.

[45] Ben Lee, A Malishevsky, D Beck, A Schmid, and E Landry. "Dynamic Branch Prediction". In: *Oregon State University* ().

[46] Jonathan Levin. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons, 2012.

[47] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security Symposium*. 2016.

[48] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *S&P*. 2015.

[49] LWN. *The current state of kernel page-table isolation*. Dec. 2017. URL: https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/.

[50] Giorgi Maisuradze and Christian Rossow. "Speculose: Analyzing the Security Implications of Speculative Execution in CPUs". In: *arXiv:1801.04084* (2018).

[51] Tarjei Mandt. *Attacking the iOS Kernel: A Look at 'evasi0n'*. 2013. URL: www.nislab.no/content/download/38610/481190/file/NISlecture201303.pdf.

[52] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: *NDSS*. 2017.

[53] Matt Miller. *Mitigating speculative execution side channel hardware vulnerabilities*. Mar. 2018. URL: https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/.

[54] Ingor Molnar. *x86: Enable KASLR by default*. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6807c84652b0b7e2e198e50a9ad47ef41b236e59.

[55] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES". In: *CT-RSA*. 2006.

[56] Colin Percival. "Cache missing for fun and profit". In: *BSDCan*. 2005.

[57] Phoronix. *Linux 4.12 To Enable KASLR By Default*. 2017. URL: https://www.phoronix.com/scan.php?page=news_item&px=KASLR-Default-Linux-4.12.

[58] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks". In: *NDSS*. 2018.

[59] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 2011.

[60] Elvira Teran, Zhe Wang, and Daniel A Jiménez. "Perceptron learning for reuse prediction". In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–12.

[61] Robert M Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[62] Lucian N Vintan and Mihaela Iridon. "Towards a high performance neural branch predictor". In: *Neural Networks, 1999. IJCNN'99. International Joint Conference on*. Vol. 2. IEEE. 1999, pp. 868–873.

[63] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014.

[64] Tse-Yu Yeh and Yale N Patt. "Two-level adaptive training branch prediction". In: *Proceedings of the 24th annual international symposium on Microarchitecture*. ACM. 1991, pp. 51–61.

[65] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-Tenant Side-Channel Attacks in PaaS Clouds". In: *CCS*. 2014.

Appendix

9.1. Meltdown in Practice

In this section, we show how Meltdown can be used in practice. In Section 9.1.1, we show physical memory dumps obtained via Meltdown, including passwords of the Firefox password manager. In Section 9.1.2, we demonstrate a real-world exploit.

9.1.1. Physical-memory Dump using Meltdown

listing 6.3 shows a memory dump using Meltdown on an Intel Core i7-6700K running Ubuntu 16.10 with the Linux kernel 4.8.0. In this example, we can identify HTTP headers of a request to a web server running on the machine. The `XX` cases represent bytes where the side channel did not yield any results, *i.e.*, no *Flush+Reload* hit. Additional repetitions of the attack may still be able to read these bytes.

Listing 6.4 shows a memory dump of Firefox 56 using Meltdown on the same machine. We can clearly identify some of the passwords that are stored in the internal password manager, *i.e.*, `Dolphin18`, `insta_0203`, and `secretpwd0`. The attack also recovered a URL which appears to be related to a Firefox add-on.

9.1.2. Real-world Meltdown Exploit

In this section, we present a real-world exploit showing the applicability of Meltdown in practice, implemented by Pavel Boldin in collaboration with Raphael Carvalho. The exploit dumps the memory of a specific process, provided either the process id (PID) or the process name.

First, the exploit de-randomizes the kernel address space layout to be able to access internal kernel structures. Second, the kernel's task list is traversed until the victim process is found. Finally, the root of the victim's multilevel page table is extracted from the task structure and traversed to dump any of the victim's pages.

The three steps of the exploit are combined to an end-to-end exploit which targets a specific kernel build and a specific victim. The exploit can easily be adapted to work on any kernel build. The only requirement is access to either the binary or the symbol table of the kernel, which is true for all public kernels which are distributed as packages, *i.e.*, not self-compiled. In the remainder of this section, we provide a detailed explanation of the three steps.

Breaking KASLR. The first step is to de-randomize KASLR to access internal kernel structures. The exploit locates a known value inside the kernel, specifically the Linux banner string, as the content is known and it is large enough to rule out false positives. It starts looking for the banner string at the (non-randomized) default address according to the

```
79cbb80: 6c4c 48 32 5a 78 66 56 44  73 4b 57 39 34 68 6d |lLH2ZxfVDsKW94hm|
79cbb90: 3364 2f 41 4d 41 45 44 41  41 41 41 41 51 45 42 |3d/AMAEDAAAAAQEB|
79cbba0: 4141 41 41 41 3d 3d XX  XX XX XX XX XX XX XX |AAAAAA==........|
79cbbb0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbc0: XXXX XX 65 2d 68 65 61 64  XX XX XX XX XX XX XX |...e-head.......|
79cbbd0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbe0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbf0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc00: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc10: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc20: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc30: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc40: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc50: XXXX XX XX 0d 0a XX 6f 72  69 67 69 6e 61 6c 2d |.......original-|
79cbc60: 7265 73 70 6f 6e 73 65 2d  68 65 61 64 65 72 73 |response-headers|
79cbc70: XX44 61 74 65 3a 20 53 61  74 2c 20 30 39 20 44 |.Date: Sat, 09 D|
79cbc80: 6563 20 32 30 31 37 20 32  32 3a 32 39 3a 32 35 |ec 2017 22:29:25|
79cbc90: 2047 4d 54 0d 0a 43 6f 6e  74 65 6e 74 2d 4c 65 | GMT..Content-Le|
79cbca0: 6e67 74 68 3a 20 31 0d 0a  43 6f 6e 74 65 6e 74 |ngth: 1..Content|
79cbcb0: 2d54 79 70 65 3a 20 74 65  78 74 2f 68 74 6d 6c |-Type: text/html|
79cbcc0: 3b20 63 68 61 72 73 65 74  3d 75 74 66 2d 38 0d |; charset=utf-8.|
```

Listing (6.3) Memory dump showing HTTP Headers on Ubuntu 16.10 on a Intel Core i7-6700K

```
f94b76f0: 12 XX e0 81 19 XX e0 81 44  6f 6c 70 68 69 6e 31 |........Dolphin1|
f94b7700: 38 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |8...............|
f94b7710: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX |pR.k............|
f94b7720: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b7730: XX XX XX XX 4a XX XX XX XX  XX XX XX XX XX XX |....J...........|
f94b7740: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b7750: XX XX XX XX XX XX XX XX XX  XX e0 81 69 6e 73 74 |............inst|
f94b7760: 61 5f 30 32 30 33 e5 e5 e5  e5 e5 e5 e5 e5 e5 |a_0203..........|
f94b7770: 70 52 18 7d 28 7f XX XX XX  XX XX XX XX XX XX |pR.}(...........|
f94b7780: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b7790: XX XX XX XX 54 XX XX XX XX  XX XX XX XX XX XX |....T...........|
f94b77a0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b77b0: XX XX XX XX XX XX XX XX XX  XX XX XX 73 65 63 72 |............secr|
f94b77c0: 65 74 70 77 64 30 e5 e5 e5  e5 e5 e5 e5 e5 e5 |etpwd0..........|
f94b77d0: 30 b4 18 7d 28 7f XX XX XX  XX XX XX XX XX XX |0..}(...........|
f94b77e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b77f0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b7800: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 |................|
f94b7810: 68 74 74 70 73 3a 2f 2f 61  64 64 6f 6e 73 2e 63 |https://addons.c|
f94b7820: 64 6e 2e 6d 6f 7a 69 6c 6c  61 2e 6e 65 74 2f 75 |dn.mozilla.net/u|
f94b7830: 73 65 72 2d 6d 65 64 69 61  2f 61 64 64 6f 6e 5f |ser-media/addon_|
```

Listing (6.4) Memory dump of Firefox 56 on Ubuntu 16.10 on a Intel Core i7-6700K disclosing saved passwords.

symbol table of the running kernel. If the string is not found, the next attempt is made at the next possible randomized address until the target is found. As the Linux KASLR implementation only has an entropy of 6 bits [37], there are only 64 possible randomization offsets, making this approach practical.

The difference between the found address and the non-randomized base address is then the randomization offset of the kernel address space. The remainder of this section assumes that addresses are already de-randomized using the detected offset.

Locating the Victim Process. Linux manages all processes (including their hierarchy) in a linked list. The head of this task list is stored in the `init_task` structure, which is at a fixed offset that only varies among different kernel builds. Thus, knowledge of the kernel build is sufficient to locate the task list.

Among other members, each task list structure contains a pointer to the next element in the task list as well as a task's PID, name, and the root of the multilevel page table. Thus, the exploit traverses the task list until the victim process is found.

Dumping the Victim Process. The root of the multilevel page table is extracted from the victim's task list entry. The page table entries on all levels are physical page addresses. Meltdown can read these addresses via the direct-physical map, *i.e.*, by adding the base address of the direct-physical map to the physical addresses. This base address is `0xffff 8800 0000 0000` if the direct-physical map is not randomized. If the direct-physical map is randomized, it can be extracted from the kernel's `page_offset_base` variable.

Starting at the root of the victim's multilevel page table, the exploit can simply traverse the levels down to the lowest level. For a specific address of the victim, the exploit uses the paging structures to resolve the respective physical address and read the content of this physical address via the direct-physical map. The exploit can also be easily extended to enumerate all pages belonging to the victim process, and then dump any (or all) of these pages.

7

Nethammer: Inducing Rowhammer Faults through Network Requests

Publication Data

Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. "Nethammer: Inducing Rowhammer Faults through Network Requests". In: *SILM Workshop*. 2020

Contributions

Main author.

Nethammer: Inducing Rowhammer Faults through Network Requests

Moritz Lipp[1], Michael Schwarz[1], Lukas Raab[1], Lukas Lamster[1], Misiker Tadesse Aga[2], Clémentine Maurice[3], Daniel Gruss[1]

[1] Graz University of Technology [2] Univ Rennes, CNRS, IRISA [3] University of Michigan

Abstract

In this paper, we present Nethammer, a remote Rowhammer attack without a single attacker-controlled line of code on the targeted system, *i.e.*, not even JavaScript. Nethammer works on commodity consumer-grade systems that either are protected with quality-of-service techniques like Intel CAT or that use uncached memory, flush instructions, or non-temporal instructions while handling network requests (e.g., for interaction with the network device). We demonstrate that the frequency of the cache misses is in all three cases high enough to induce bit flips. Our evaluation showed that depending on the location, the bit flip compromises either the security and integrity of the system and the data of its users, or it can leave persistent damage on the system, *i.e.*, persistent denial of service. We invalidate threat models of Rowhammer defenses building upon the assumption of a local attacker. Consequently, we show that most state-of-the-art defenses do not affect our attack. In particular, we demonstrate that target-row-refresh (TRR) implemented in DDR4 has no aggravating effect on local or remote Rowhammer attacks.

1. Introduction

Hardware-fault attacks have been considered a security threat since at least 1997 [6, 7]. In such attacks, the attacker intentionally brings devices into physical conditions that are outside their specification for a short time. For instance, this can be achieved by temporarily using incorrect supply voltages, exposing them to high or low temperatures, exposing them to radiation, or by dismantling the chip and shooting at it with lasers. Fault attacks typically require physical access to the device. However, if software can bring the device to the border or outside

of the specified operational conditions, software-induced hardware faults are possible [34, 51].

The Rowhammer bug is a hardware reliability issue of DRAM [34]. An attacker can exploit this bug by repeatedly accessing (*hammering*) DRAM cells at a high frequency, causing unauthorized changes in physically adjacent memory locations. Examples of Rowhammer attacks include privilege escalation from native environments [17, 49], from within a browser's sandbox [18], and from within virtual machines running on third-party compute clouds [55], mounting fault attacks on cryptographic primitives [5, 48], and obtaining root privileges on mobile phones [54].

Intel CAT is a quality-of-service feature [21], allowing to restrict cache allocation of cores to a subset of cache ways of the last-level cache, removing interference of workloads in shared environments, e.g., protecting virtual machines against performance degradation due to cache thrashing of co-located virtual machines. However, Aga et al. [1] showed that Intel CAT facilitates eviction-based Rowhammer attacks.

The large majority of previous Rowhammer attacks required some form of local code execution, e.g., JavaScript [18] or native code [1, 4, 5, 17, 34, 41, 46, 48, 49, 54, 55]. Consequently, all works on Rowhammer defenses assume that some form of local code execution is required [4, 8, 9, 16, 20, 27, 33, 34, 44, 58]. In contrast, Tatar et al. [52] utilized RDMA-enabled network cards to perform targeted memory accesses to specific physical addresses over a remote interface to induce bit flips.

In this paper, we challenge the requirements of remote Rowhammer attacks. We present Nethammer, a Rowhammer attack that does not require local code execution, nor RDMA-enabled network cards. Nethammer only requires a fast network connection between the attacker and the victim. It sends a crafted stream of size-optimized packets to the victim, causing a high number of memory accesses to the same set of memory locations. If any software processing the network request (e.g., user application, shared libraries, network stack, network driver) use uncached memory, non-temporal instructions or flush instructions (e.g., for interaction with the network device) an attacker can induce bit flips. Furthermore, if Intel CAT is activated, e.g., as an anti-DoS mechanism, memory accesses lead to fast cache eviction and, thus, frequent DRAM accesses, *i.e.*, Rowhammer. While, as in the first practical Rowhammer attacks [49], an attacker cannot control the addresses of the bit flips, we demonstrate how an attacker can still exploit them and reduce the

probability of flips in non-attacker controlled regions by spraying.

To build Nethammer, we systematically analyzed the requirements to induce bit flips and, in particular, real-world memory-controller page policies. In most Rowhammer attacks, two DRAM rows are hammered to induce bit flips. The reason is that they assume that an "open-page" memory controller policy is used, *i.e.*, a DRAM row is kept open until a different row is accessed. However, modern CPUs employ more sophisticated memory controller policies that preemptively close rows [17]. We demonstrate one-location hammering [17] with adaptive page policies for the first time.

We also analyzed memory operations during network requests and analyzed the Nethammer bit flips we empirically obtained on our target systems and different potential target applications. In all cases, the triggered bit flips may induce persistent denial-of-service attacks by corrupting the persistent state, e.g., the file system on the remote machine. We empirically observed bit flips using Nethammer already after 300 ms runtime and up to 10 000 per hour.

Finally, we evaluate state-of-the-art defenses and show that most of them do not affect our attack. In particular, we show that TRR does not mitigate Rowhammer.

Contributions. The contributions of this work are:

- We present Nethammer, a remote Rowhammer attack that does not require attacker-controlled code on the target device, nor RDMA-enabled network cards.
- We demonstrate Nethammer on systems using uncached memory (or `clflush`) while handling network packets.
- We show how memory controller policies can automatically be identified.
- We show that the TRR countermeasure in DDR4 has no significant effect on Rowhammer attacks.

Outline. Section 2 provides background. Section 3 gives an attack overview. Section 4 describes the building blocks. Section 5 describes

specific exploit strategies. Section 6 evaluates our empiric results. In Section 7, we discuss limitations and specific defenses. Section 8 concludes.

Responsible Disclosure. We responsibly informed Intel about Nethammer on March 20, 2018. We disclosed a full report of Nethammer, including the ineffectiveness of TRR on DDR4 to Intel, ARM, Qualcomm, on May 11, 2018.

2. Background and Related Work

In this section, we discuss background information and related work on DRAM, memory controller policies, and the Rowhammer attack. Furthermore, we discuss caches and cache eviction as well as the Intel CAT technology.

DRAM and Memory Controller Policies. DRAM in modern computers is organized for a high degree of parallelism, in a hierarchy of 1–4 channels, one or more DIMMs, 1–4 ranks, 1–4 bank groups, and 8 or 16 banks. Each bank is an array of *cells*, organized in *rows* and *columns*, storing the actual memory content. The memory controller translates physical addresses to channel, DIMM, rank, bank group, bank, row, and column addresses. Pessl et al. [45] reverse-engineered these addressing functions using an automated technique for several processors.

As DRAM cells lose their charge over time, they must be refreshed periodically. The refresh interval is defined as 64 ms but can be adjusted to compensate, e.g., for temperature.

Each bank has a *row buffer*, buffering any read and write accesses to rows in this bank. Hence, depending on the state of the row buffer three different cases can occur: Row hits are the fastest, an access to a row in a pre-charged bank (*i.e.*, no row in the row buffer) is a few nanoseconds slower, row conflicts (*i.e.*, other row in row buffer) are measurably slower. The memory controller can optimize the memory performance by deciding when to close a row preemptively and pre-charge the bank. Typically, memory controllers employ one of the three following page policies:

1. *Closed-page policy*: the page is immediately closed, and the bank is pre-charged.

2. *Fixed open-page policy*: the page is left open for a fixed amount of time. This policy is beneficial for high-locality workloads, for power consumption and bank utilization [32].

3. *Adaptive open-page policy*: the adaptive open-page policy by Intel [12] is similar to the fixed open-page policy but dynamically adjusts the page timeout interval per bank.

As modern processors have many cores running independently as well as deploy large caches and complex algorithms for spatial and temporal prefetching, the probability that subsequent memory accesses go to the same row decreases. Awashti et al. [3] proposed an access-based page policy that assumes a row receives the same number of accesses as the last time it was activated. Shen et al. [50] proposed a policy taking past memory accesses into account to decide whether to close a row preemptively. Intel suggested predicting how long a row should be kept open [31, 53]. Consequently, more complex memory controller policies have been proposed and are implemented in modern processors [16, 32].

Rowhammer. Increasing DRAM cell density achieves higher storage capacity and lower power consumption, but cells may be more susceptible to disturbance errors [41], *i.e.*, bit flips. Such bit flips can be induced from software by bypassing the cache using specific instructions [34], cache eviction [1, 4, 14, 18], or uncached memory [46, 54]. Different access patterns have been developed to induce Rowhammer bit flips:

1. *Single-sided hammering* [49] accesses 8 randomly chosen memory locations simultaneously. The probability is high that at least 2 out of 8 random memory locations map into the same out of 32 DRAM banks on DDR3.

2. *Double-sided hammering* hammers two rows sandwiching a third. This requires at least partial knowledge of virtual-to-physical and physical-to-DRAM mappings.

3. *One-location hammering* [17] only accesses one single location at a high frequency. The attacker does not directly induce row conflicts but instead keeps re-opening one row permanently. As modern processors do not use strict open-page policies anymore, the memory controller preemptively closes rows earlier than necessary, causing row conflicts on the subsequent accesses of the attacker.

Using these techniques, the Rowhammer bug has been exploited in different scenarios and environments, e.g., attacking [5], sandboxes [14, 18, 49], native environments [17, 49], virtual machines [48, 55], mobile devices [54].

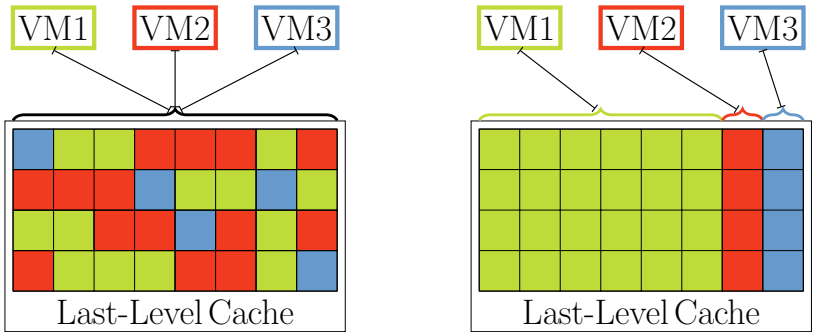
To develop defenses, a large body of research focused on detecting [9, 19, 20, 27, 44, 58], neutralizing [8, 18, 48, 54], or eliminating [4, 8, 16, 33, 34] Rowhammer attacks in software or hardware. The NethammerLPDDR4 standard [30] specifies two features to mitigate Rowhammer attacks: with Target Row Refresh (TRR) the memory controller refreshes adjacent rows of a certain row and with Maximum Activation Count (MAC) the number of times a row can be activated before adjacent rows have to be refreshed is specified. One-location hammering however bypasses all software-based defenses [17].

Tatar et al. [52] utilized RDMA-enabled network cards to induce bit flips remotely. RDMA enables remote access to specific physical addresses in a controlled way and, hence, can be used to implement Rowhammer memory access patterns. RDMA-enabled network cards are expensive and are only used by a few cloud providers [39]. In 2019, Cojocar et al. [10] demonstrated Rowhammer attacks bypassing ECC protection. In March 2020, Frigo et al. [15] analyzed TRR in more depth, confirming our findings of Section 6.

Caches and Cache Eviction. Hardware caches keep frequently used data from main memory in smaller but faster memories. Modern CPUs have multiple cache levels, with the L3 cache usually being the largest but slowest cache, shared across cores and inclusive to lower-level caches. The L3 cache on such CPUs has sets consisting of a fixed number of cache ways, where the set is determined by the physical address, and a replacement policy decides which way to replace (evict).

To mount a Rowhammer attack, an attacker needs to bypass the cache, e.g., via the unprivileged `clflush` instruction [56], or uncached memory [54]. An attacker can also resort to cache eviction by accessing congruent memory addresses [14, 18, 35], *i.e.*, addresses that map to the same cache set. Gruss et al. [18] observed that it is important to trick the replacement policy into keeping memory locations of the attacker cached, rather than the victim address that the attacker wants to evict.

In 2016, Intel introduced Cache Allocation Technology (CAT) [25] to address quality of service in multi-core server platforms [21, 24]. Intel CAT



(a) CAT disabled (b) CAT enabled

Figure 7.1.: When Intel CAT is disabled in (a), the cache is shared among the virtual machines. In (b), CAT is configured with 6 ways for VM1, and 1 way for VM2 and VM3.

allows system software to partition the last-level cache to optimize workloads in shared environments as well as to isolate applications or virtual machines on servers. When a virtual machine on a server thrashes the cache and therefore decreases the performance of other co-located machines, the hypervisor can restrict this virtual machine to a subset of the cache to retain the performance of other tenants. More specifically, Intel CAT allows restricting the number of cache ways available to processes, virtual machines, and containers, as illustrated in Figure 7.1. However, Aga et al. [1] showed that Intel CAT allows improving eviction-based Rowhammer attacks as it reduces the number of accesses, and thus the time, required for cache eviction.

3. Nethammer Attack

In this section, we present Nethammer, a Rowhammer attack not relying on any attacker-controlled code on the victim machine, nor RDMA-enabled network cards.

Attack Overview. Nethammer sends a crafted stream of network packets to the target device to mount a one-location or single-sided Rowhammer attack. For each packet received on the target device, a set of addresses is accessed, e.g., in the kernel driver, in a user-space application processing the contents, somewhere in between (e.g., network stack,

shared libraries), or a combination of all. By repeatedly sending packets, this set of addresses is hammered and, thus, bit flips may be induced. As frequently-used addresses are served from the cache for performance, the cache must be bypassed such that the access goes directly into the DRAM to cause the row conflicts required for hammering. This can be achieved in different ways if the code that is executed (in kernel space or user space) when receiving a packet,

1. *evicts* (and later on reloads) an address;
2. uses *uncached* memory;
3. uses *non-temporal* instructions;
4. *flushes* (and later on reloads) an address.

Non-temporal instructions perform their operations directly to the memory bypassing the cache [46]. *Uncached* memory is used on virtually all ARM-based devices for interaction with the hardware, e.g., access buffers used by the network controller. Intel x86 processors have the `clflush` instruction for the same purpose, and we found several open-source repositories where the `clflush` instruction was used for interaction with the hardware, but only one of them was an (outdated) network driver. We still describe this attack for completeness' sake, as it also applies to closed source drivers or user-space applications that handle the received packets and possibly use the `clflush` instruction. We verified that an attack is practical in both scenarios, as we describe in Section 6.

However, the main focus and contribution of this paper is an eviction-based remote Rowhammer attack. As caches are large and cache replacement policies try to keep frequently-used data in the cache, it is not trivial to mount an eviction-based attack without executing attacker-controlled code on the device. However, to address quality of service in multi-core server platforms, Intel introduced CAT (cf. Section 2), allowing to control the amount of cache available to applications or virtual machines dynamically, as illustrated in Figure 7.1. If a virtual machine is thrashing the cache, the hypervisor limits the number of cache ways available to this virtual machine to meet performance guarantees given to other tenants on the same host. Thus, if an attacker excessively uses the cache, its virtual machine is restricted to a low number of ways, possibly only one, leading to a fast self-eviction of addresses.

To induce bit flips remotely, one requirement is to send as many packets as possible over the network in a short time frame. As an example,

UDP packets without content can be used, allowing an overall packet size of 64 B, which is the minimum packet size for an Ethernet packet. This allows to send up to 1 024 000 packets per second over a 500 Mbit/s connection.

Attack Setup. In our attack setup, the attacker has a fast network connection to the victim machine, e.g., a gigabit connection. We assume that the victim machine has DDR2, DDR3, or DDR4 memory that is susceptible to one-location (or single-sided) hammering. As DRAM with ECC can detect and correct single-bit errors and, thus, complicates Rowhammer attacks, we assume non-ECC memory on the victim machine. We did find server systems that have no ECC memory in the wild [22, 23, 28, 42]. Note that this is not a real limitation, as Cojocar et al. [10] demonstrated Rowhammer attacks bypassing ECC protection. Furthermore, we assume that the victim machine uses either Intel CAT, available in Xeon CPUs, or uncached memory while handling network packets. We found 12.7 % of the dedicated hosts for sale on Hetzner [22] to have a Xeon CPU but non-ECC memory. Finally, we assume that the attacker has a sufficiently fast network connection to the victim, see Section 4.3. For our attack on personal computers, tablets, smartphones, or devices with similar hardware configuration, we make no further assumptions.

4. From Regular Memory Accesses to Rowhammer

We investigate memory-controller page policies to determine whether regular memory accesses that occur while handling network packets could, at least in theory, induce bit flips. Note that these investigations are oblivious to the specific technique to access the DRAM row (*i.e.*, eviction, flushing, uncached memory).

In Section 4.1, we propose a method to determine the memory-controller page policy on real-world systems automatically. We show that one-location hammering does not necessarily need a closed-page policy, but instead, adaptive policies may allow one-location hammering.

Based on these insights, we demonstrate the first one-location Rowhammer attack on an ARM device in Section 4.2. Finally, we investigate whether Rowhammer via network packets is theoretically possible. This is not trivial, as network packets do not arrive at the same speed as the memory accesses in an optimized tight loop.

4.1. Automated Classification of Memory-Controller Page Policies

Gruss et al. [17] found that the memory-controller page policy has a significant influence on the way the Rowhammer bug can be triggered. In particular, they found that one-location hammering works and deduced from this that the memory-controller page policy must be similar to a closed-page policy. To get a more in-depth understanding of the memory-controller page policy used on a specific system, we present an automated method to detect the used policy. This is a significant step forward for Rowhammer attacks, as it allows to deduce whether specific attack variants may or may not work without an empiric evaluation.

The undocumented mapping functions [45] allow to select addresses to access specific DRAM channels, ranks, banks, but also rows. Accessing same-bank different-row addresses consecutively causes a row conflict in the corresponding bank, incurring higher latency for the second access as the currently active row must be closed (written back), the bank must be pre-charged, and only then the new row can be fetched with an activate command.

We assume knowledge of processor and DRAM timings. For the DRAM this means in particular, the `tRCD` latency (the time to select a column address), and the `tRP` latency (the time between pre-charge and row activation). These three timings influence the observed latency as follows:

1. we consider the case **page open / row hit** as the baseline;
2. in the case **page empty / bank pre-charged**, we observe an additional latency of `tRP` over a row hit;
3. in the case **page miss / row conflict**, we observe an additional latency of (`tRP` + `tRCD`) over a row hit.

To compute the actual number of cycles we can expect, we have to divide the DRAM latency value by the DRAM clock rate. In the case of DDR4, we have to additionally divide the latency value by factor two, as DDR4 is double-clocked. This yields the latency in nanoseconds. By dividing the nanoseconds by the processor clock speed, we obtain the latency in CPU cycles. Still, as we cannot obtain absolutely clean measurements due to out-of-order execution, prefetching, and other mechanisms that aim to hide the DRAM latency, the actually observed latency will deviate slightly.

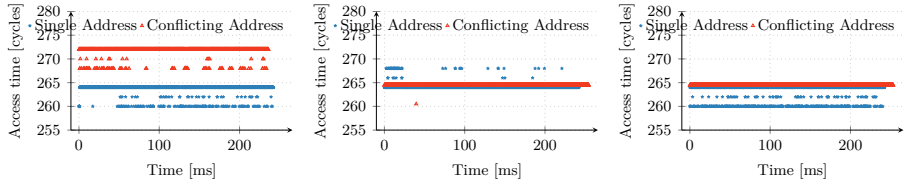
As in our test we cannot simply measure the three different cases, we define an experiment that allows to distinguish the different policies. In the experiment we use for our automated classification, we select two addresses A and B that map to the same bank but different rows. Using the `clflush` instruction, we make sure that A and B are not cached, in order to load those addresses directly from main memory. We base our method on two observations for open-page policies:

- **Single**: By loading address A an increasing number of times ($n = 1..10\,000$) before measuring the time it takes to load the same address on a subsequent access, we can measure the access time of an address in DRAM if the corresponding row is already active. For an open-page policy, the access time should be the same for any n.
- **Conflict**: By accessing address A and subsequently measuring the access time to address B, we can measure the access time of an address in DRAM in the occurrence of a row conflict.

Our classification runs the following checks:

1. If there is no timing difference between the two cases described above (**Single** with a large n and **Conflict**), the system uses a closed-page policy. The closed-page policy immediately closes the row after every read or write request. Thus, there is no timing difference between these two cases. The timing observed corresponds to the row-pre-charged state.
2. Otherwise, if the timing for the **Single** case is the same, regardless of the value of n, but differs from the timing for **Conflict**, the system uses an open-page policy. The timing difference corresponds to the row hits and conflicts. Following the definition of the open-page policy, the timing for row hits is always the same.
3. Otherwise, the timing for the **Single** case will have a jump at some n, after which the page policy is adapted to cope better with our workload. Consequently, the timing differences we observe correspond to row hit and row-pre-charged states.

Figure 7.2 shows the memory access time measured on an Intel Xeon D-1541 with different page policies, *i.e.*, the closed-page policy can be distinguished using our method. We also verified our results by reading out the `CLOSE_PG` bit in the `mcmtr` configuration register of the integrated memory controller [26].



(a) Open-page policy (b) Closed-page policy (c) Adaptive policy

Figure 7.2.: Measured access times over a period of time for a single address (blue) and an address causing a row conflict (red) for different page policies on the Intel Xeon D-1541: open policy (left), closed policy (middle), adaptive policy (right).

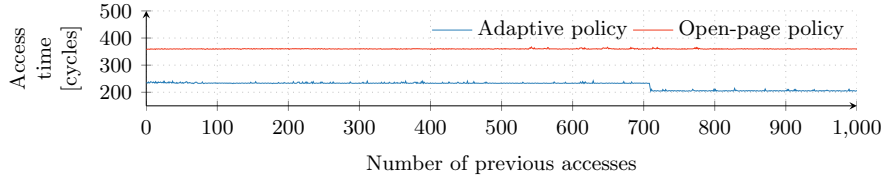


Figure 7.3.: Open-page policy and adaptive page policy can be distinguished by testing increasing numbers of accesses to the same row.

We validated that we can distinguish open-page policy and adaptive page policy by running our experiments on two systems with the corresponding page policies. Figure 7.3 shows the results of these experiments. The difference between open-page policy and adaptive policy is clearly visible.

Our experiments show that adaptive page policies often behave like closed-page policies. This indicates the possibility of one-locating hammering on systems using an adaptive page policy.

4.2. One-location Hammering on ARM

To make Nethammer a more generic attack, it is essential to demonstrate it not only on Intel CPUs but also on ARM CPUs. This is particularly interesting as ARM CPUs dominate the mobile market, and ARM-based devices are predominant also in IoT applications. Gruss et al. [17] only demonstrated one-location hammering on Intel CPUs. However, as one-location hammering is the most plausible hammering variant for Nethammer, we need to investigate whether it is possible to trigger one-location

hammering bit flips on ARM.

In our experiments, we used a LG Nexus 4 E960 mobile phone equipped with a Qualcomm Snapdragon 600 (APQ8064) SoC and 2GB of LPDDR2 RAM, susceptible to bit flips using double-sided hammering. The page policy used by the memory controller is selected via the `DDR_CMD_EXEC_OPT_0` register: if the bit is set to 1 (the recommended value [47]), a closed-page policy is used. If the bit is set to 0, an open-page policy is used. Hence, we can expect the memory controller to preemptively close rows, enabling one-location hammering.

So far, bit flips on ARM-based devices have only been demonstrated in the combination of double-sided hammering, and uncached memory [54], or access via the GPU [14]. Even in the presence of a flush instruction [2] or optimal cache eviction strategies [35], the access frequency to the two neighboring rows is too low to induce bit flips. Furthermore, devices with the ARMv8 instruction set that allows exposing a flush instruction to unprivileged programs are usually equipped with NethammerLPDDR4 memory.

In our experiment, we allocated uncached memory using the Android ION memory allocator [57]. We hammered a single random address within the uncached memory region at a high frequency and then checked the memory for occurred bit flips. We were able to observe 4 bit flips while hammering for 10 hours. Thus, we can conclude that there are ARM-based devices that are vulnerable to one-location hammering.

4.3. Minimal Access Frequency for Rowhammer Attacks

Nethammer requires a high frequency of memory accesses caused by processing network packets. Previous work indicated that at least 43 000 to 139 000 row activations [4, 18, 34] are required within one refresh interval to induce a bit flip.

In our experiments, we send 500 Mbit/s (and more) over the network interface. With a minimum size of 64 B for ethernet packets, this corresponds to 1 024 000 packets per second. Several kernel functions are called multiple times, e.g., up to 6 times (cf. Section 6). Hence, on a 500 Mbit/s connection, the attack can induce 6 144 000 accesses per second. Divided by the default refresh interval of 64 ms, we are at 393 216 accesses per refresh interval. This is clearly above the previously reported required number of memory accesses [4, 18, 34]. Hence, we conclude that in the-

ory, if the system is susceptible to Rowhammer attacks, network packets can induce bit flips. In the following section, we will describe how an attacker can exploit such bit flips.

5. Exploiting Bit Flips over a Network

Nethammer does not control where in physical memory a bit flip is induced and, thus, what is stored at that location, the bit flip can have different consequences. We distinguish between bit flips in user memory, *i.e.*, memory pages that are mapped as `user_accessible` in at least one process and bit flips in kernel memory. We can also distinguish the bit flips based on their high-level effect, again forming two groups, depending on whether or not they lead to a denial-of-service situation. A denial-of-service situation can be persistent if the bit flip is written back to a permanent storage location. Then it may be necessary to reinstall the system software or parts of it from scratch, clearly taking more time than just a reboot.

File System Data Structures. File system data structures, e.g., inodes, are not directly part of the kernel code or data but are also in the kernel memory. An inode is a data structure defining a file or a directory of a file system. Each inode contains metadata, such as the size of the file, owner, and permission data, as well as the disk block location of its data. If a bit flips in the inode structure, it corrupts the file system and, thus, causes persistent loss of data. This may again crash the entire system. We empirically validated that this case occurs.[1]

SGX Enclave Page Cache. Bit flips in this region lock up the memory controller instantly (unsafely), halting the entire system [17, 29]. We empirically validated that this case occurs and found unsafe halting of the system to often leave permanent file system damage.

Application Memory. If a bit flip occurs in a user-space application, e.g., code or data, a possible outcome is the crash of the program. Such a flip may render the affected service unavailable. Another outcome of a bit flip in the data of a user-space application, e.g., in the database of

[1] In fact, it was a problem when trying to trigger the other cases.

a service, is that the service delivers modified, possibly invalid, content. Depending on the service, its users cannot distinguish if the data is correct or has been altered.

One example are DNS entries, which are altered such that a character of DNS entry points to a different domain, *i.e.*, bitsquatting [11]. Such bit flips in domains have been successfully exploited before using Rowhammer attacks [48]. Using zone transfers, an attacker can retrieve entries of an entire zone. The attacker queries the DNS server for its entries, mounts the attack, and then verifies whether a bit flip at an exploitable position has occurred by monitoring changes in the queried entries. If so, the attacker can register the changed domain and host a malicious service on the domain, e.g., a phishing website or a mail server intercepting email traffic. Users querying the DNS server for said entry connect to the attacker-controlled server and are thus exposed to data theft.

An attacker can also target Online Certificate Status Protocol (OCSP) servers that allow querying the status of a single certificate. The OCSP server manages a list of revoked certificate fingerprints. Liu et al. [36] evaluated 74 full IPv4 HTTPS scans and found that 8% of 38 514 130 unique SSL certificates served have been revoked. The attacker flips a bit in the memory of an OCSP server of a certificate authority where private keys of certificates have become public, and the certificates have thus been revoked. The attacker can either flip the status or the identifier of the certificate (a chance of 99.875 % *per bit flip* in the OCSP revocation list). A bit flip in the certificate identifier leads to the OCSP server not finding the certificate in its database anymore, thus, returning "unknown" as the state. Most browsers fall back to their own certificate revocation list in such a case [43]. However, only high-value revocations are kept in the browser's list, making it very unlikely that the certificate is in the certificate revocation list of the browser. Hence, an attacker can again reuse that certificate.

We empirically observed bit flips in these applications, with a lower frequency than denial-of-service bit flips.

Cryptographic Material. Cryptographic material as part of the application memory is particularly interesting for attacks. To commit changes to a version-controlled repository, users authenticate with the service using public-key cryptography. As the position of the bit flip cannot be controlled using Nethammer, an attacker can improve the probability to

induce a bit flip in the modulus of a public key by loading as many keys as possible into the main memory of the server. Some APIs, e.g., the GitLab API, allow enumerating the registered users as well as their public keys. By enumerating and, thus, accessing all public keys of the service, the attacker loads the public keys into the DRAM. In the first step of the attack, the attacker enumerates all keys of all users and stores them locally. In the second step, the attacker mounts Nethammer to induce bit flips on the targeted system. The more keys the attacker loaded into memory, the more likely it is that the bit flip corrupts the modulus of a public key of a user. For instance, with 80 % of the memory filled with 4096-bit keys, the chance to hit a bit of a modulus is 79.7 %. As the attacker does not know which key was affected by the bit flip, the attacker enumerates all keys again and compares them with the locally stored keys. If a modified key has been found, the attacker computes a new corresponding private key [40, 48]. Consequently, the attacker can make changes to the software repository as that user and, thus, introduce bugs that can later be exploited if the software is distributed. The original public key is restored after a while when the key is evicted from the page cache and reloaded from the hard drive. As the correct key is restored, the attack leaves no traces. Furthermore, it breaks the non-repudiation guarantee provided by the public-key authentication, making the victim whose public key was attacked the prime suspect in possible investigations.

6. Evaluation

In this section, we evaluate Nethammer and its performance. We show that the number of bit flips induced by Nethammer depends on how the cache is bypassed and the memory controller's page policy. We evaluate which kernel functions are executed when handling a UDP network packet. We describe the bit flips we obtained when running Nethammer in different attack scenarios. Finally, we show that TRR, a countermeasure against Rowhammer implemented in some DDR4 RAMs, does not protect against Nethammer or Rowhammer in general.

Environment. In our evaluation, we used the test systems listed in Table 7.1. We used the second and third system for our experiments with Intel CAT, which was configured exactly as recommended for quality-of-service purposes. For completeness' sake, on the first system, we ran an unprivileged server application which uses `clflush` while handling re-

Table 7.1.: List of test systems that were used for the experiments.

Device	CPU	DRAM	Network card	Operating system
Desktop	Intel i7-6700K @ 4 GHz	8 GB DDR4 @ 2133 MHz	Intel 10G X550T	Ubuntu 16.04
Server	Intel Xeon E5-1630v4 @ 3.7 GHz	8 GB DDR4 @ 2133 MHz	Intel i210/i218-LM Gigabit	Xubuntu 17.10
Server	Intel Xeon D-1541 @ 2.1 GHz	8 GB DDR4 @ 2133 MHz	Intel i350-AM2 Gigabit	Ubuntu 16.04
LG Nexus 4	Qualcomm APQ8064 @ 1.5 GHz	2 GB LPDDR2 @ 533 MHz	USB Adapter	Android 5.1.1

quests, and in another experiment, we installed a network driver which uses `clflush` while interacting with the network card. To mount Nethammer, we used a Gigabit switch to connect two other machines with the victim machine. The two other machines were used to flood the victim machine with network packets triggering the Rowhammer bug. We used the fourth system for our experiments on an ARM-based device that uses uncached memory in the process of handling a network packet.

Evaluation of the Different Cache Bypasses for Nethammer. In Section 4, we investigated the requirements to trigger the Rowhammer bug over the network. In this section, we evaluate Nethammer for three cache-bypass techniques (cf. Section 3): Intel Xeon CPUs with Intel CAT for fast cache eviction, uncached memory on an ARM-based mobile device, and a single `clflush` instruction in the code running when receiving a packet.

The operating system will handle every network packet received by the network card. The operating system parses the packets depending on their type, validates their checksum and copies, and delivers every packet to each registered socket queue. Thus, for each received packet, quite some code is executed before the packet finally arrives at the application destined to handle its content.

We tested Nethammer on Intel Xeon CPUs with Intel CAT. The number of cache ways has been limited to a single one for code handling the processing of UDP packets, resulting in fast cache eviction. If a function is called multiple times for one packet, the same addresses are accessed and loaded from DRAM with a high probability, thus, hammering this location. To estimate how many different functions are called and how often they are called, we use the *perf* framework to count the number of function calls related to UDP packet handling. Section A shows the results of a system handling UDP packets. Out of 27 different functions we identified, most were called only once for each received packet. The function `__udp4_lib_lookup` is called twice. In a more extensive scan, we

found that `nf_hook_slow` is called 6 times while handling UDP packets on some kernels.

With this knowledge, we analyzed how many bit flips can be induced by this code execution. We observed 45 bit flips per hour on the Intel Xeon E5-1630v4. As TRR is active on this system (see Section 6), fewer bit flips occur in comparison to systems without TRR. In Section 6, we evaluate the number of bit flips depending on the configured page policy.

In Section 4.2, we demonstrated that ARM-based devices are vulnerable to one-location hammering in general. To investigate whether bit flips can also be induced over the network, we connect the LG Nexus 4 using an OTG USB ethernet adapter to a local network. Using a different machine, we send as many network packets as possible to the mobile phone. An application on the phone allocates memory and repeatedly checks the allocated memory for occurred bit flips. However, we were not able to observe any bit flips on the device within 12 hours of hammering. As the device does not deploy technology like Intel CAT (Section 2), the cache is not limited for certain applications and, thus, the eviction caused by handling memory packets has a low probability. As network drivers often use DMA memory and, thus, uncached memory, bit flips can be induced by network packets. While we identified a remarkable number of around 5500 uncacheable pages used by the system, we were not able to induce any bit flips remotely. However, the USB ethernet adapter allowed only a network capacity of less than $16\,\mathrm{Mbit\,s^{-1}}$, which is clearly too slow for a Nethammer attack. Nevertheless, we were successfully able to induce bit flips using Nethammer on the Intel Xeon E5-1630v4, where one uncached address is accessed for every received UDP packet. Non-temporal instructions directly operate on the memory, thus, behaving similar to uncached memory.

We implemented an unprivileged userspace server application which uses `clflush` while handling network requests. We then also tested a network driver implementation that uses `clflush` in the process of handling a network packet. Both tests were performed on an Intel i7-6700K CPU. While `clflush` is not very commonly used, our experiments provide valuable insights into the implications if it is used somewhere. We sent UDP packets with up to $500\,\mathrm{Mbit\,s^{-1}}$ and scanned memory regions where we expected bit flips. The results for both cases were very similar. For the driver variant, we observed a bit flip every 350 ms showing that hammering over the network is feasible if at least two memory accesses are served from main memory, due to flushing an address while handling a network

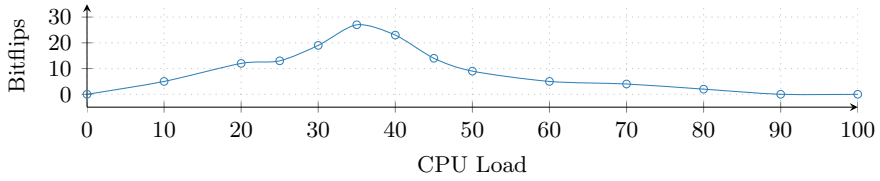


Figure 7.4.: Number of bit flips depending on the CPU load with a closed-page policy after 15 minutes.

packet. Thus, in this scenario, up to 10 000 bit flips per hour can be induced.

Influence of Memory-Controller Page Policies on Rowhammer. In order to evaluate the actual influence of the used memory-controller page policy on Nethammer, *i.e.*, how many bit flips can be induced depending on the policy used, we mounted the Nethammer in different settings. The experiment was conducted on our Intel Xeon D-1541 test system, as the BIOS of its motherboard allowed to chose between different page policies: *Auto*, *Closed*, *Open*, *Adaptive*. For each run, we configured the victim machine with one of the policies and Intel CAT, and, mounted a Nethammer attack for at least 4 hours. To detect bit flips, we ran a program on the victim machine that mapped a file into memory. The program then repeatedly scans the content of all allocated pages and reports bit flips if the content has changed.

We detected 11 bit flips in 4 hours with the *Closed* policy, with the first one after 90 minutes. We did not observe any bit flips with the *Open* policy within the first 4 hours. However, when running the experiment longer, we observed 46 bit flips within 10 hours. With the *Adaptive* policy, we observed 10 bit flips in 4 hours, with the first one within the second hour of the experiment. While this experiment was conducted without any additional load on the system, we see in Figure 7.4 that additional CPU utilization increases the number of bit flips drastically. Using the *Closed* policy, we observed 27 bit flips with a load of 35% within 15 minutes.

These results do not immediately align with the assumption that a policy that preemptively closes rows is required to induce bit flips using one-location hammering. However, depending on the addresses that are accessed and the constant eviction through Intel CAT, it is possible that

two addresses map to the same bank but different rows and, thus, bit flips can be induced through single-sided hammering. In fact, the attacker cannot know whether the hammering was actually one-location hammering or single-sided hammering. However, as long as a bit flip occurs, the attacker does not care how many addresses mapped to the same bank. Finally, depending on the actual parameters used by a fixed-open-page policy, a row can still be closed early enough to induce bit flips.

Bit Flips induced by Nethammer. As described in Section 5, a bit flip can occur in user space or kernel space leading to different effects depending on the memory it corrupts. In this section, we present bit flips that we have observed in our experiments and their effects.

We observed Nethammer bit flips that caused the system not to boot anymore. It stopped responding after the bootloader stage. We inspected the kernel image and compared it to the original kernel image distributed by the operating system. As the kernel image differed blockwise at many locations, we assume that Nethammer caused a bit flip in a file-system inode. The inode of a program that wanted to write data did not point to the correct file but to the kernel image and, thus, corrupted the kernel image.

Furthermore, we observed several bit flips immediately halting the entire system with no further interaction possible. By debugging the operating system over a serial connection, we detected bit flips in certain modules such as the keyboard or network driver. In these cases, the system was still running but did not respond to any user input or network packets anymore. We also observed bit flips that were likely in the SGX EPC region, causing an immediate permanent locking of the memory controller.

We observed that bit flips crashed running processes and services or prevented the execution of others as the bit flip triggered a segmentation fault when functions of a library were executed. On one occasion, a bit flip occurred either in the SSH daemon or the stored passwords of the machine, preventing to log in on the system. The system was restored to a stable state by rebooting the machine and thus reloading the entire code.

We also validated that an attacker can increase chances of a bit flip in a target page by increasing the memory usage. This was the most common scenario, overlapping with our test setup to detect bit flips for our eval-

uation. Unsurprisingly, these flips equally occur when filling the memory with actual content that the attacker targets.

Target Row Refresh (TRR). Previous assumptions on the Rowhammer bug lead to the conclusion that only bit flips in the victim row adjacent to the hammering rows would occur. While the probability for bit flips to occur in directly adjacent rows is much higher, Kim et al. [34] already showed rows further away (even a distance of 8 rows and more) are affected as well. Still, the hardware vendors opted for implementing defenses focusing on the directly adjacent rows.

With the Low Power Double Data Rate 4 (NethammerLPDDR4) standard, the NethammerLPDDR4 standard defines a reliability feature called Target Row Refresh (TRR). The idea of TRR is to refresh adjacent rows if the targeted row is accessed at a high frequency. More specifically, TRR works with a maximum number of activations allowed during one refresh cycle, the maximum active count. Thus, if a double-sided Rowhammer attack (Section 2) is mounted, and two hammered rows are accessed more than the defined maximum active count, the adjacent rows (in particular the victim row of the attack) will be refreshed. As the potential victim rows are refreshed, in theory, no bit flip will occur, and the attack is mitigated. However, in practice, bit flips can be further away from the hammered rows, and thus, TRR may be ineffective.

With the Ivy Bridge processor family, Intel introduced Pseudo Target Row Refresh (pTRR) for Intel Xeon CPUs to mitigate the Rowhammer bug [38]. On these systems, pTRR-compliant DIMMs must be used; otherwise, the system will default into double refresh mode, where the time interval in which a row is refreshed is halved [38]. However, Kim et al. [34] showed that a reduced refresh period of 32 ms is not sufficient enough to impede bit flips in all cases. While pTRR is implemented in the memory controller [37], DRAM module specifications allow automatically running TRR in the background.

In our experiments, we were able to induce bit flips on a pTRR-supporting DDR4 module using double-sided hammering on an Intel i7-6700K. The bit flips occurred in directly adjacent rows and rows further away. We observed that when using the same DDR4 DRAM on the Intel Xeon E5-1630 v4 CPU, no bit flips occurred in the directly adjacent rows, but we observed no statistically significant difference in the number of bit flips for the rows further away. This indicates that TRR is active on the second

machine but also that TRR does not prevent the occurrence of exploitable bit flips in practice. Thus, we conclude that the TRR hardware defense is insufficient in mitigating Rowhammer attacks. In March 2020, Frigo et al. [15] analyzed TRR in more depth, confirming our findings that TRR does not prevent Rowhammer in practice.

7. Discussion

To induce the Rowhammer bug, one needs to access memory in the main memory repeatedly and, thus, needs to circumvent the cache. Therefore, either native flush instructions [56], eviction [1, 18] uncached memory [54] or non-temporal instructions [46] can be used to remove data from the cache. In particular, for eviction-based Nethammer, the system must use Intel CAT as described in Section 2 in a configuration that restricts the number of ways available to a virtual machine in a cloud scenario to guarantee performance to other co-located machines [24]. Furthermore, the DRAM has to be susceptible to Rowhammer. We discovered in a brief market survey that many cloud providers offer hardware without ECC RAM [22, 23, 28, 42], potentially allowing Nethammer attacks.

Nethammer sends as many network packets to the victim machine as possible, aiming to induce bit flips. Depending on the actual attack scenario (cf. Section 5), additional traffic, e.g., by enumerating the public keys of the service, is generated. If the victim uses network monitoring software, the attack might be prevented due to the highly increased amount of traffic. In our experiments, we sent a stream of UDP packets with up to 500 Mbit/s to the target system. We could induce a bit flip every 350 ms. Thus, if the first random bit flip already hits the target or causes a denial-of-service, the attack could already be successful. As the rows are periodically refreshed, an attacker only needs a burst of memory accesses to a row between two refreshes, *i.e.*, within a period of 64 ms. Hence, an attacker could mount Nethammer for a few hundred milliseconds and then pause the attack for a longer time to prevent detection. While ethernet adapters in mobile phones are uncommon, many ARM-based embedded devices in IoT setups have gigabit ethernet.

The maximum throughput of these network cards we measured was too low on many of these devices, e.g., the Raspberry Pi 3 Model B+ [13], and WiFi chips typically offer too little capacity. However, on more recent modems, e.g., the Qualcomm X20 Gigabit LTE modem, throughputs up

to 1.2 Gbit/s are possible in practice. This would enable sending enough packets to hammer specific addresses and potentially induce bit flips on the device.

8. Conclusion

In this paper, we presented Nethammer, a remote Rowhammer attack, with no attacker-controlled line of code on the target system. We demonstrate attacks on commodity consumer-grade systems, leading to temporary or persistent damage to the system. In some cases, the system was rendered unbootable after the attack. Our method to automatically identify the page policy used by the memory controller allowed us to show that adaptive page policies are also vulnerable to one-location hammering. We demonstrated the first one-location hammering attack on an ARM device, indicating their future exposure to Nethammer. Finally, we demonstrated that target-row-refresh (TRR) on DDR4 memory has no aggravating effect on local or remote Rowhammer attacks.

Acknowledgments

We thank our reviewers for their comments and suggestions that helped improving the paper. The project was supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria, and Carinthia. It was also supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work also benefited from the support of the project ANR-19-CE39-0007 MIAOUS of the French National Research Agency (ANR). Additional funding was provided by generous gifts from Intel and Red Hat. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. "When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks". In: *HOST*. 2017.

[2] ARM. *ARM Architecture Reference Manual ARMv8*. ARM Limited, 2013.

[3] Manu Awasthi, David W. Nellans, Rajeev Balasubramonian, and Al Davis. "Prediction Based DRAM Row-Buffer Management in the Many-Core Era". In: *PACT*. 2011.

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. "ANVIL: Software-based protection against next-generation Rowhammer attacks". In: *ACM SIGPLAN Notices* (2016).

[5] Sarani Bhattacharya and Debdeep Mukhopadhyay. "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis". In: *CHES*. 2016.

[6] Eli Biham. "A fast new DES implementation in software". In: *International Workshop on Fast Software Encryption*. 1997.

[7] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults". In: *EUROCRYPT*. 1997.

[8] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. "CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory". In: *USENIX Security Symposium*. 2017.

[9] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. *Real time detection of cache-based side-channel attacks using Hardware Performance Counters*. ePrint 2015/1034. 2015.

[10] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks". In: *S&P*. 2019.

[11] Artem Dinaburg. "Bitsquatting: DNS Hijacking without Exploitation". In: *BlackHat US Briefings*. 2011.

[12] James M. Dodd. *Adaptive page management*. 2003.

[13] Raspberry Pi Foundation. *Raspberry Pi 3 Model B+*. 2018. URL: `https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus`.

[14] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU". In: *S&P*. 2018.

[15] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "TRRespass: Exploiting the Many Sides of Target Row Refresh". In: *S&P*. 2020.

[16] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. *ARMOR: A Run-time Memory Hot-Row Detector*. 2015. URL: `http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer`.

[17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. "Another Flip in the Wall of Rowhammer Defenses". In: *S&P*. 2018.

[18] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *DIMVA*. 2016.

[19] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *DIMVA*. 2016.

[20] Nishad Herath and Anders Fogh. "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security". In: *BlackHat US Briefings*. 2015.

[21] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. "Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family". In: *IEEE HPCA*. 2016.

[22] Hetzner. *Dedicated Root Server Hosting*. 2018. URL: `https://www.hetzner.com/dedicated-rootserver/`.

[23] DefineQuality Hosting. *Highend Dedicated Rootserver*. 2018. URL: `https://definequality.net/dedicated.php`.

[24] Intel. *Improving Real-Time Performance by Utilizing Cache Allocation Technology: Enhancing Performance via Allocation of the Processor's Cache*. 2015. URL: `https://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html`.

[25] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2019.

[26] Intel. *Intel Xeon Processor E5 v4 Product Family: Datasheet Volume 2: Registers*. 2016.

[27] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *MASCAT: Stopping Microarchitectural Attacks Before Execution*. ePrint 2016/1196. 2017.

[28] myLoc managed IT. *The dedicated server in comparison*. 2018. URL: https://www.myloc.de/en/server-hosting/dedicated-server/dedicated-server-comparison.html.

[29] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. "SGX-Bomb: Locking Down the Processor via Rowhammer Attack". In: *SysTEX*. 2017.

[30] JEDEC Solid State Technology Association. *Low Power Double Data Rate 4*. 2017. URL: http://www.jedec.org/standards-documents/docs/jesd209-4b.

[31] Suryaprasad Kareenahalli, Zohar B. Bogin, and Mihir D. Shah. *Adaptive idle timer for a memory device*. 2003.

[32] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era". In: *MICRO*. 2011.

[33] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. "Architectural support for mitigating row hammering in DRAM memories". In: *IEEE Computer Architecture Letters* 14 (2015).

[34] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *ISCA*. 2014.

[35] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security Symposium*. 2016.

[36] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. "An End-to-End Measurement of Certificate Revocation in the Web's PKI". In: *IMC*. 2015.

[37] Sreenivas Mandava, Brian S. Morris, Suneeta Sah, Roy M. Stevens, Ted Rossin, Mathew W. Stefaniw, and John H. Crawford. *Techniques for determining victim row addresses in a volatile memory*. 2017.

[38] Marcin Kaczmarski. *Thoughts on Intel Xeon E5-2600 v2 Product Family Performance Optimisation – component selection guidelines*. Infobazy 2014. Aug. 2014. URL: http://infobazy.gda.pl/2014/pliki/prezentacje/d2s2e4-Kaczmarski-Optymalna.pdf.

[39] Microsoft Azure. *High performance compute VM sizes*. 2018. URL: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-hpc.

[40] James A Muir. "Seifert's RSA fault attack: Simplified analysis and generalizations". In: *International Conference on Information and Communications Security*. 2006.

[41] Onur Mutlu. "The RowHammer problem and other issues we may face as memory becomes denser". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2017.

[42] netcup. *Dedicated servers for professional applications*. 2018. URL: https://www.netcup.eu/professional/dedizierte-server/.

[43] Paul Mutton. *Certificate revocation: Why browsers remain affected by Heartbleed*. 2014. URL: https://news.netcraft.com/archives/2014/04/24/certificate-revocation-why-browsers-remain-affected-by-heartbleed.html.

[44] Matthias Payer. "HexPADS: a platform to detect "stealth" attacks". In: *ESSoS*. 2016.

[45] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *USENIX Security Symposium*. 2016.

[46] Rui Qiao and Mark Seaborn. "A New Approach for Rowhammer Attacks". In: *International Symposium on Hardware Oriented Security and Trust*. 2016.

[47] Inc. Qualcomm Technologies. *Qualcomm Snapdragon 600E Processor APQ8064E: Recommended Memory Controller and Device Settings*. 2016.

[48] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. "Flip Feng Shui: Hammering a Needle in the Software Stack". In: *USENIX Security Symposium*. 2016.

[49] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges". In: *Black Hat Briefings*. 2015.

[50] X. Shen, F. Song, H. Meng, S. An, and Z. Zhang. "RBPP: A row based DRAM page policy for the many-core era". In: *IEEE ICPADS*. 2014.

[51] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *USENIX Security Symposium*. 2017.

[52] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Throwhammer:

Rowhammer Attacks over the Network and Defenses". In: *USENIX ATC*. 2018.

[53] Chee Hak Teh, Suryaprasad Kareenahalli, and Zohar Bogin. *Dynamic update adaptive idle timer*. 2006.

[54] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms". In: *CCS*. 2016.

[55] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation". In: *USENIX Security Symposium*. 2016.

[56] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014.

[57] Thomas M. Zeng. *The Android ION memory allocator*. 2012. URL: `https://lwn.net/Articles/480055/`.

[58] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. "CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds". In: *RAID*. 2016.

Appendix

A. Kernel Accesses for Network Packets

Table 7.2 shows the results of the *funccount* script of the *perf* framework for functions with udp in their name while the targeted system is flooded with UDP packets.

Table 7.2.: Results of *funccount* on the victim machine for functions with `udp` in their name while the system is flooded with UDP packets.

Function	Number of calls
`__udp4_lib_lookup`	2 000 024
`__udp4_lib_rcv`	1 000 012
`udp4_gro_receive`	1 000 012
`udp4_lib_lookup_skb`	1 000 012
`udp_error`	1 000 012
`udp_get_timeouts`	1 000 013
`udp_gro_receive`	1 000 013
`udp_packet`	1 000 012
`udp_pkt_to_tuple`	1 000 012
`udp_rcv`	1 000 012
`udp_v4_early_demux`	1 000 012

8

Practical Keystroke Timing Attacks in Sandboxed JavaScript

Publication Data

Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. "Practical Keystroke Timing Attacks in Sandboxed JavaScript". In: *ESORICS*. 2017

Contributions

Main author.

Practical Keystroke Timing Attacks in Sandboxed JavaScript

Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, Stefan Mangard

Graz University of Technology

Abstract

Keystrokes trigger interrupts which can be detected through software side channels to reconstruct keystroke timings. Keystroke timing attacks use these side channels to infer typed words, passphrases, or create user fingerprints. While keystroke timing attacks are considered harmful, they typically require native code execution to exploit the side channels and, thus, may not be practical in many scenarios.

In this paper, we present the first generic keystroke timing attack in sandboxed JavaScript, targeting arbitrary other tabs, processes and programs. This violates same-origin policy, HTTPS security model, and process isolation. Our attack is based on the interrupt-timing side channel which has previously only been exploited using native code. In contrast to previous attacks, we do not require the victim to run a malicious binary or interact with the malicious website. Instead, our attack runs in a background tab, possibly in a minimized browser window, displaying a malicious online advertisement. We show that we can observe the exact inter-keystroke timings for a user's PIN or password, infer URLs entered by the user, and distinguish different users time-sharing a computer. Our attack works on personal computers, laptops and smartphones, with different operating systems and browsers. As a solution against all known JavaScript timing attacks, we propose a fine-grained permission model.

1. Introduction

Keystroke timing attacks are side-channel attacks where an adversary tries to determine the exact timestamps of user key presses. Keystroke timings convey sensitive information that has been exploited in previous work to recover words and sentences [39, 49]. More recently, microarchitectural attacks have been demonstrated to obtain keystroke timings [15,

25, 32, 36] in native code. In particular, the interrupt-timing side channel leaks highly accurate keystroke timings if an adversary has access to a cycle-accurate timing source [36].

JavaScript is the most widely used scripting language and supported by virtually any browser today. It is commonly used to create interactive website elements and enrich the user interface. However, it does not provide access to native instructions, files, or system services. Still, the ability to execute arbitrary code in the JavaScript sandbox inside a website can also be exploited to perform attacks on website visitors, e.g., timing attacks [12].

JavaScript-based timing attacks were first presented by Felten et al. [12], showing that access times to website elements are lower if a website has recently been visited. Besides attacks on the browser history [12, 21, 46], there have also been more fine-grained attacks recovering information on the user or other websites visited by the user [8, 16, 22, 40, 41]. Vila and Köpf [43] showed that shared event loops in Google Chrome leak timing information on other browser tabs that share worker processes responsible for rendering or I/O.

Previous work has shown that timing side channels which are introduced on the hardware level or the operating system level, can be exploited from JavaScript. Gruss et al. [14] demonstrated page deduplication attacks, Oren et al. [30] demonstrated cache attacks to infer mouse movements and network activity, and Booth [6] fingerprinted websites based on CPU utilization. Gras et al. [13] showed that accurate timing information in JavaScript can be exploited to defeat address-space layout randomization. Schwarz et al. [37] presented a DRAM timing covert channel in JavaScript.

In this paper, we present the first generic keystroke timing attack in sandboxed JavaScript. Our attack is based on the interrupt-timing side channel which has previously only been exploited using native code. We show that this side channel can be exploited from JavaScript without access to native instructions. Based on instruction throughput variations within equally-sized time windows, we can detect hardware interrupts, such as keyboard inputs. In contrast to previous side-channel attacks in JavaScript, our channel provides a more accurate signal for keystrokes, allowing us to observe exact inter-keystroke timings. We demonstrate how this information can be used to infer URLs entered by the user, and distinguish different users time-sharing a computer.

Our attack is generic and can be applied to any system which uses in-

terrupts for user input. We show that our attack code works both on personal computers and laptops, as well as modern smartphones. An adversary can target other browser tabs and browser processes, as well as arbitrary other programs, circumventing same-origin policy, HTTPS security model, and both operating system and browser-level process isolation. With a low impact on the overall system and browser performance, and a code footprint of less than 256 bytes of code, the attack can easily be hidden in modern JavaScript frameworks and malicious online advertisements. Our attack code utilizes new JavaScript features to run in the background, in a background tab, or on a locked phone. Hence, we can spy on the PIN entry used to unlock the phone.

To verify our results, we implemented our attack also in Java without access to native instructions and only low-accuracy timers. We demonstrate that the same timing measurements as in JavaScript can be observed in our Java implementation with a lower noise level. Furthermore, we demonstrate that in a cross-browser covert channel two websites can communicate through network interrupts. These observations clearly show that the source of the throughput differences is caused by the hardware and not specific software implementations.

Our attack works in two phases, an online phase running in JavaScript, and an offline phase running on the adversary's machine. In the offline phase, we employ machine learning techniques to build accurate classifiers trained on keystroke traces gathered in the online phase. These classifiers enable an adversary to infer which website a victim opens and to fingerprint different users time-sharing the same physical machine (e.g., a family sharing a computer).

Our results show that side-channel attacks are a fundamental problem that is not restricted to local adversaries. We propose a fine-grained permission model as a solution against all known JavaScript timing attacks. The browser restricts access to specific features and prompts the user to grant permissions per domain.

Our key contributions are:

- We show the first generic keystroke timing attack in JavaScript, embedded in a website, targeting arbitrary other tabs, processes and programs.
- We demonstrate our attack on personal computers, laptops and smartphones, with different browsers and operating systems.

- We demonstrate that our attack can obtain the exact inter-keystroke timings for a user's PIN or password, infer URLs entered by the user, and distinguish different users time-sharing a computer based on their input.

Outline. The remaining paper is organized as follows. In Section 2, we provide background. We describe our attack in Section 3. In Section 4, we present the performance of our attack on personal computers and smartphones. We discuss countermeasures in Section 5. Finally, we conclude in Section 6.

2. Background

2.1. Keystroke Timing Attacks

Keystroke timing attacks acquire accurate timestamps of keystrokes for input sequences. These keystroke timestamps depend on several factors such as bigrams, syllables, words, keyboard layout, and typing experience [33]. An adversary can exploit these timing characteristics to learn information about the user or the user input. Existing attacks use machine learning to infer typed sentences or recover passphrases [38, 39, 49]. Idrus et al. [19] showed that key press and key release events can be used to fingerprint users.

The Linux operating system exposes information that allows compiling accurate traces of keystroke timings [39, 49]. Zhang et al. [49] demonstrated that instruction and stack pointer, interrupt statistics, and network packet statistics can be used as side channels for keystroke timings. While Song et al. [39] demonstrated that SSH leaks inter-keystroke timings in interactive mode, Hogye et al. [17] showed that network latency in networks with significant traffic conceals these inter-keystroke timings in practice. Kamran et al. [3] showed that it is possible to detect keystrokes and classify the typed keys using Wi-Fi Signals. Jana and Shmatikov [20] showed that CPU usage is a much more reliable side channel for keystroke timings than the instruction pointer, or the stack pointer. Diao et al. [11] demonstrated high-precision keystroke timing attacks based on `/proc/interrupts`. Mehrnezhad et al. [27] used the JavaScript sensor API to detect touch, hold, scroll, and zoom actions on mobile devices using built-in sensors such as accelerometer and gyroscope.

Algorithm 1: Online phase of an interrupt-timing attack

```
input  : threshold
now ← get_timestamp();
while true do
    last ← now;
    now ← get_timestamp();
    if now − last > threshold then
        report(now, diff);
    end
end
```

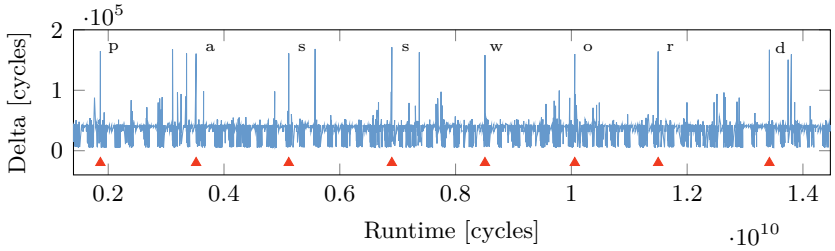


Figure 8.1.: Native interrupt-timing attack: The difference between consecutive timestamps is measured while a sentence is typed. Every keystroke leads to a significant deviation as the measuring program is interrupted by the keyboard.

Cache attacks have also been used to obtain keystroke timings. In a cache attack, the adversary observes effects of the victim's operation on the cache and can then deduce what operations the victim performed. Ristenpart et al. [34] demonstrated a keystroke timing attack using a *Prime+Probe* cache attack. Gruss et al. [15] demonstrated that *Flush+Reload* cache attacks can be used for keystroke timing attacks.Similarly, Pessl et al. [32] showed a keystroke timing attack on the Firefox address bar using the DRAM as a side channel.

Recently, it was shown that keystroke interrupt timings can be obtained in a timing attack which continuously measures differences between consecutive `rdtsc` calls [36]. However, this is not possible if the adversary only controls a website that is visited by the victim. Sandboxed JavaScript running on a website cannot utilize any native instructions such as `rdtsc`.

2.2. Interrupt-timing Attacks

Interrupt-timing attacks have recently been demonstrated in native code to recover keystroke timings [36]. The basic idea of interrupt-timing attacks is to continuously acquire a high-resolution timestamp and to monitor differences between subsequent timestamps, *i.e.*, how much time has passed since the last measurement, as outlined in Algorithm 1. Significant differences occur whenever the measuring process is interrupted. The more time the operating system consumes to handle the interrupt, the higher the measured differences are. Especially interrupts triggered by I/O devices—such as keyboards—lead to clearly visible peaks in the measured trace. Figure 8.1 shows a trace from a native attack implementation while a user typed in a sentence. The exact timestamp where the user pressed a key is clearly visible and can be distinguished from other events. However, the trace does not only contain keyboard interrupts and, thus, allows spying on user input but also on every other event that causes one or more interrupts, e.g., network traffic or redraw events. An adversary can filter relevant peaks by means of post-processing algorithms to monitor entered keystrokes.

2.3. Timing Attacks in Sandboxed JavaScript

JavaScript has evolved to be the most widely supported scripting language, notably because it is supported by virtually every modern browser. With highly-optimized just-in-time compilation, modern JavaScript engines deliver a performance that can compete with native code implementations. The timestamp counter provides a cycle-accurate timestamp to user programs in native code, but it is not accessible from JavaScript. Instead, JavaScript provides the High Resolution Time API [44] (`performance.now`) for sub-millisecond timestamps.

Based on this timing interface, various attacks have been demonstrated. Van Goethem et al. [41] were able to extract private data from users by measuring the differences in the execution time from cross-origin resources. Stone [40] showed that the optimization in SVG filters introduced timing side channels. He showed that this side channel can be used to extract pixel information from iframes. Booth [6] fingerprinted websites based on CPU utilization—interfering with the execution time of a benchmark function—when loading and rendering the page.

Gruss et al. [14] showed that page deduplication timing attacks can be

performed in JavaScript to determine which websites the user has currently opened. Oren et al. [30] showed that it is possible to mount cache attacks in JavaScript. They demonstrated how to perform *Prime+Probe* attacks in the browser to build cache covert channels but also to spy on the user's mouse movements and network activity through the cache. This attack caused all major browsers to decrease the resolution of the `performance.now` method [1, 7, 10]. The W3C standard now recommends a resolution of 5 µs while the Tor project reduced the resolution in the Tor browser to a more conservative value of 100 ms [28]. Gras et al. [13] showed that accurate timing information in JavaScript can be exploited to defeat address-space layout randomization. Vila and Köpf [43] showed that shared event loops in Google Chrome leak timing information about other browser tabs sharing worker processes for rendering and I/O operations. They exploit this side channel to identify web pages, to build a covert communication channel, and to infer inter-keystroke timings.

Recently, several works investigated timing primitives in JavaScript that allow recovering highly accurate timestamps [13, 24, 37]. We use these timing primitives to build highly accurate keystroke timing attacks in sandboxed JavaScript.

3. Sandboxed Keystroke Timing Attacks without High-resolution Timers

Our attack follows the same idea as interrupt-timing attacks in native code [36]. It consists of an online phase where timing traces are acquired on a victim machine and an offline phase for post-processing and evaluation.

Online phase. In the online phase of our attack, we run an interrupt-timing attack in sandboxed JavaScript. Interrupt-timing attacks have only minimal requirements, most importantly access to the x86 `rdtsc` instruction [36]. Consequently, keystroke interrupt-timing attacks have only been demonstrated in native code. We face several challenges to perform keystroke interrupt-timing attacks from remote websites, as JavaScript can neither execute this instruction nor run endless loops on websites.

There is no high-resolution timestamp available in JavaScript, as the resolution of `performance.now` is limited to 5 µs to mitigate side-channel

attacks [44]. Therefore, we implement a counter to simulate a monotonic clock by constantly incrementing a value [13, 24, 37, 47]. The number of increments, *i.e.*, the instruction throughput, is proportional to the time the counter function is scheduled. Thus, any interrupt reduces the instruction throughput and, therefore, leads to a lower number of increments within a fixed time frame. Consequently, we can read the counter value at fixed time intervals and deduce from the number of increments since the last interval whether the counter function was interrupted.

As JavaScript is based on a single-threaded event loop, browsers usually do not allow websites to use endless loops and inform the user when detecting such a construct. The usual solution is to either use `setTimeout` or `setInterval` to constantly trigger execution of the loop body after a specified number of milliseconds have passed. However, these functions enforce a minimum pause of 4 ms before scheduling the same code again, yielding a resolution that is significantly lower than the resolution of `performance.now`.

To work around this limitation, we introduce a new variant of previously published timing primitives [13, 24, 37] called cooperative endless-loop slicing. The idea is to slice the endless loop into smaller finite loops where every loop slice has an execution time of approximately 4 ms. Before running this loop, we schedule the next loop slice using `setTimeout` with a timeout of 4 ms. Thus, in the optimal case, the next slice of the endless loop is executed immediately after the current slice, giving the impression of an actual endless loop. However, as higher priority events, such as user inputs, can still be processed between the loop slices, the browser is responsive and will not stop the endless loop. Algorithm 2 illustrates how we use this construct to continuously schedule our counter to obtain continuous timing traces.

The instruction throughput per loop slice, *i.e.*, the counter increments, varies depending on how often and how long the thread was interrupted during this loop slice. Within one loop slice, we achieve on average 72 764 increments of the counter, resulting in a resolution of approximately 69 ns ($\sigma = 3$ ns, $n = 4000$) on an Intel i5-6200U. This resolution is three orders of magnitude higher than the result of Vila and Köpf [43] who achieved a resolution of only 25 µs to 100 µs. On ARM, we achieve on average 5038 increments on the Google Nexus 5 and 17 454 increments on the OnePlus 3T, yielding a resolution of 994 ns ($\sigma = 55$ ns, $n = 4000$) and 287 ns ($\sigma = 4$ ns, $n = 4000$) respectively.

Algorithm 2: Interrupt-timing attack implemented in JavaScript

```
Function measure_time(id):
    setTimeout(measure_time, 0, id + 1);
    counter ← 0;
    begin ← window.performance.now();
    while (window.performance.now() - begin) < 5 do
        counter ← counter + 1;
    end
    publish(id, counter);
```

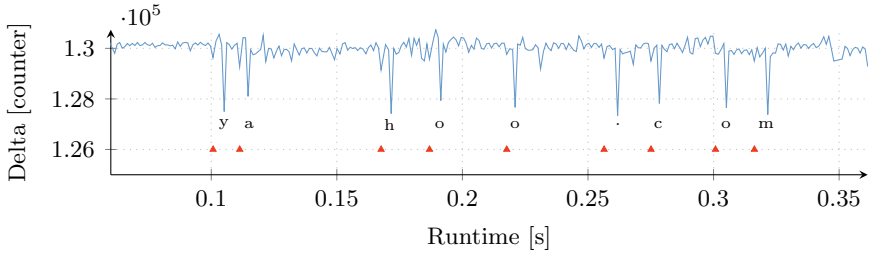


Figure 8.2.: Interrupt-timing attack in JavaScript: The lower peaks indicate that the measured script has been interrupted, allowing to infer single keystrokes.

A further limitation of JavaScript is that once the user switches the tab or minimizes the browser, the default minimum timeout value of 4 ms is reduced to 1000 ms. Increasing the loop slices to 1000 ms is not practical since it would make the browser unresponsive again. In order to circumvent this issue, we utilize the Web Worker API which explicitly allows JavaScript code to be executed in the background [45]. We discovered that the minimum timeout is not reduced for web workers and we can still measure interrupt timings with a high frequency. This allows us to monitor keystrokes when the victim is visiting a different page or even a different application.

Figure 8.2 shows a measured trace while a user typed the URL `yahoo.com` into the browser bar. If no interrupt occurs, the counter variable has been incremented for the full time window of 4 ms, defining the baseline. If an interrupt disrupts the measuring JavaScript, the counter variable is not incremented as often in the same time window, yielding to downward-facing peaks. Thus, the typed letters leave clear marks in the measured

trace, which allows inferring single keystrokes.

Offline phase. In the offline phase of the attack, the measurements gathered from the online phase are processed and analyzed. Over time, an adversary can gather thousands of traces in order to learn about the individual typing behavior of the victim or to derive an entered passphrase or PIN code. Depending on the goal of the adversary, different methods to evaluate the gathered data can be applied. In order to detect single keystrokes in a measured trace, we filter the measured trace in order to reduce noise and to deduce threshold values for keystrokes by manually inspecting one recorded trace of the target device. Using this threshold, we can further reduce the number of points in recorded traces to a minimum and, thus, increase the performance of further computations. We build a classifier by calculating the correlation between our training set and the queried trace. In order to classify entered words, we need to take into account that the points in time where a character has been entered can vary in time in our trace. Therefore, we use k-nearest neighbors (k-NN) classification [4] and calculate the correlation of the trace with every other trace in the training set using different alignments. We chose the alignment that yields the highest correlation and decide on the class giving the best match. While more computational expensive methods working with time series [5, 35] to build classifiers exist [9, 23, 48], we show that the features of the recorded measurements are strong enough such that also simpler techniques allow to build an efficient and accurate classifier.

4. Practical Attacks and Evaluation

In this section, we demonstrate the significant attack potential of our JavaScript interrupt-timing attack. Our attack does not depend on any specific browser or operating system and can therefore be performed on personal computers, laptops and smartphones. We show that it is possible to infer which website a user has entered into the browser's address bar and to profile different users sharing the same computer. Furthermore, we show that the attack can be utilized to obtain the exact timings of every digit of the PIN that is used to unlock the phone while the attack code is executed in the web browser running in the background.

4.1. URL Classification

In our first experiment, we demonstrate that using our JavaScript keystroke timing attack on a personal computer in combination with machine learning techniques, we can infer URLs that a user has entered into the address bar of the browser. We train a classifier to successfully label measurement traces of user input sequences for the URLs of the top 10 most visited websites [2]. For this experiment we used an Intel i7-6700K CPU and Firefox 52.0 running on Linux.

Every single trace consists of timestamps with a corresponding counter value (cf. Section 3) and the corresponding URL. As there are small timing variations when the user starts typing the URL and whenever the user pressed a key, the length of the trace as well as the position of the features, *i.e.*, the characteristics in the measured values describing a key stroke, within the trace varies. Thus, we need to build our classifier in a way that overcomes those difficulties. In a preparation step, we determine the maximum trace length as well as the timestamp resolution. The resolution can be obtained from the greatest common divisor of all measured timestamps of all samples. Finally, we create a linear interpolation of every sample based on the actual resolution.

The classifier assigns a class label to an unlabeled trace where each class corresponds to one URL that we train our classifier with. In order to classify a new trace, we compute the correlation of the new trace with a fixed number of randomly chosen samples for every class. As the timestamps where the user started entering the URL vary, we need to compute the correlation of two traces for different alignments. Thus, we shift one trace within a fixed time window back and forth in order to find an alignment where the correlation reaches its maximum. The average of the five highest correlations for each class decides which class the trace belongs to, *i.e.*, we choose the highest average correlation.

We evaluate our classifier by using k-fold cross-validation. We first randomly draw 20 samples as *training set* from our collected 100 measurements from every class. We then test the classifier on a randomly drawn set of the remaining 800 samples (80 per class), the *test set*. We cross-validate our classifier by performing this evaluation multiple times with randomly selected training sets.

Figure 8.3 shows the confusion matrix. Every cell shows the probability that the classifier labels a sample of a class specified by the row into a certain class specified by the column. We can clearly see that for every

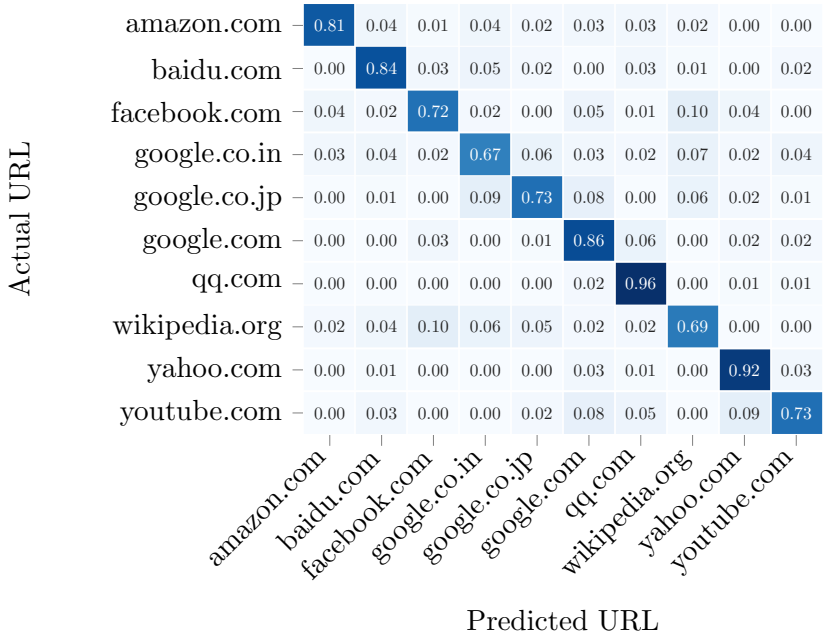


Figure 8.3.: Confusion matrix for URL input. The user input can be correctly predicted with a probability of 67 % in the worst case and 96 % in the best case. The probability of random guessing is 10 %.

domain the classifier proposes the correct class with a higher probability than an incorrect one, and a significantly higher probability than random guessing (10 %). The identification rate of *qq.com* in comparison with other domains is also very high as the domain contains only a small number of characters to be typed. The overall identification rate of our classifier is 81.75 %.

4.2. User Classification

As a second experiment, we evaluate whether it is possible to distinguish different users in order to determine who is actually sitting in front of the personal computer. In order to do so, we have collected only 5 traces of the input of the top 10 most visited websites [2] of 4 different persons to train the classifier. The results with 2 training set and 3 test set traces for each user are illustrated as a confusion matrix in Figure 8.4. While it is much harder to determine the user responsible for the given trace, our

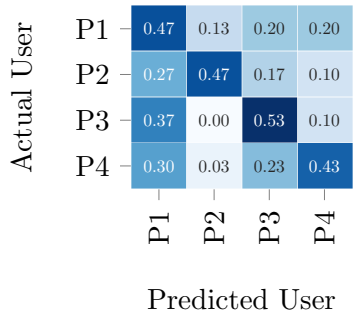


Figure 8.4.: Confusion matrix for input by different users. The user can be correctly predicted with a probability of 43 % in the worst case and 53 % in the best case. The probability of random guessing is 25 %.

Table 8.1.: Mobile test devices.

Device	SoC	Keystrokes	Screen lock
Google Nexus 5	Qualcomm MSM8974 Snapdragon 800	✓	-
Xiaomi Redmi Note 3	Mediatek MT6795 Helio X10	✓	✓
Homtom HT3	MediaTek MTK6580	✓	✓
Samsung Galaxy S6	Samsung Exynos 7420	-	✓
OnePlus One	Qualcomm MSM8974AC Snapdragon 801	✓	✓
OnePlus 3T	Qualcomm MSM8996 Snapdragon 821	-	-

classifier is with an overall identification rate of 47.5 % still better than random guessing.

4.3. Touchscreen Interactions

In our third experiment we show that interrupt-timing attacks also work on modern smartphones and on different web browsers. Although battery saving techniques should make attacks harder, the attack can still be applied if the measuring program is executed in a different tab or if the browser app is running in background. Furthermore, we show that the attack can be used to detect when the screen is locked and unlocked.

Mobile phones usually use a soft-keyboard that is displayed on the screen. Every tap on the screen causes a redraw event that is clearly visible in the measured trace, making it easier to detect when a user touches the screen. While the redraw event is sufficient to monitor taps on the keyboard, we want to be able to identify any tap on the device, whether it causes a redraw event or not. Therefore, our test website implemented a custom

touch area imitating a PIN pad.This touch area does neither register any events nor does it change its appearance. Thus, a touch onto this PIN pad should not issue any event at all, eliminating all events from the trace that are not caused by the touch interrupt itself. We provide the code for this experiment online.[1]

To cross-check whether we actually observe hardware events and not some browser-internal events, we implemented the same interrupt-detection algorithm in a native Android app. To achieve comparable results for the recorded traces, we reduced the timer resolution to 5 µs in the same way as Firefox and Chrome.

For our experiments, we used a Google Nexus 5 with a Qualcomm MSM8974 Snapdragon 800 SoC running Android 6.0.1 with Chrome 44.0.2403.133 and Firefox 54.0a1. Our second testing device is a Xiaomi Redmi Note 3 with a Mediatek MT6795 Helio X10 running Android 5.0.2 with Chrome 57.0.2987.132 and Firefox 52.0.2. In addition, we used all the device listed in Table 8.1 to record traces using the JavaScript implementation for visual inspection. Table 8.1 also shows whether we could detect keystrokes and screen locks without machine learning just by visual inspection.

Keystroke detection. Figure 8.5 shows the keystroke timing attack in a native Android app on a Google Nexus 5 where a user tapped the screen twice, before swiping once and tapping it again. The individual interrupts, caused by tapping on the phone, can easily be identified by the two following peaks representing the touch and release event. If the user swipes over the screen, many interrupts are triggered, one for every coordinate change. This results in many visible peaks and, thus, swipes and taps can be distinguished.

Our JavaScript implementation of the keystroke timing attack runs successfully in Chrome and allows distinguishing taps from swipes as illustrated in Figure 8.6. While in contrast to the native implementation, the measurements in JavaScript contain much more noise, the exact tap timings can easily be extracted and allow further, more sophisticated attacks.

Figure 8.7 shows the same trace of two taps, one swipe and one additional tap on the Xiaomi Redmi Note 3. Surprisingly, the peaks caused by the interrupts face upwards instead of downwards as one might expect. We

[1] `https://github.com/IAIK/interruptjs`

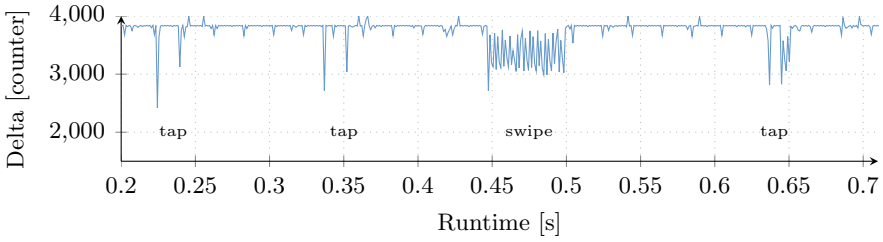


Figure 8.5.: Keystroke timing attack running in a native app on the Google Nexus 5.

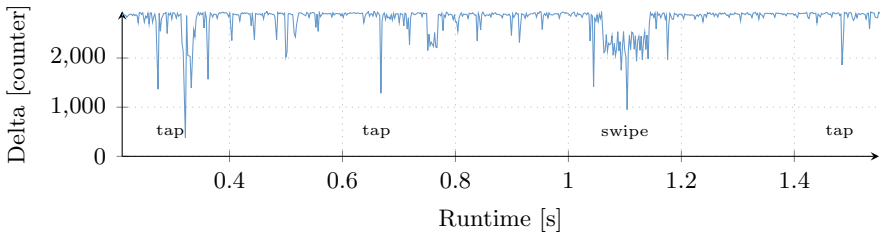


Figure 8.6.: Keystroke timing attack running in Chrome on the Google Nexus 5.

observed that the Xiaomi Redmi Note 3 increases the CPU frequency whenever the screen is touched. Consequently, although the interrupt will consume some CPU time, the counter as described in Section 3 can be incremented more often due to the significantly higher CPU frequency. We have verified this behavior by running a benchmark suite on the Xiaomi Redmi Note 3. The benchmark suite has been up to 30 % faster, when swiping over the screen while the benchmark is executed. While this feature may be useful to handle touch interrupts more efficiently and to appear more responsive, it also opens a new side channel and allows detecting tap and screen events easily. We also verify the same behavior in our native Java implementation with higher peaks which allows detecting tap and swipe events even more reliably. On the OnePlus 3T we were not able to detect keystrokes at all. We suspect that this is due to the big.LITTLE architecture, which moves the CPU-intensive browser task to a high-performance ARM core, while the interrupts are handled by smaller cores. Thus, the browser is not interrupted if a hardware interrupt occurs.

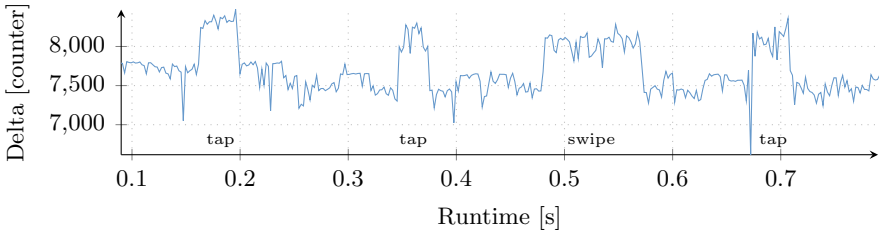


Figure 8.7.: Keystroke timing attack running in Chrome on the Xiaomi Redmi Note 3. The peaks face upwards instead of downwards as with other devices.

Spying on other applications and PIN unlock. While the attack of Vila and Köpf [43] is limited to spy on tabs or pop-ups opened by the adversary, our attack is not restricted and can be used to monitor any other application running on the system. Indeed, the attack of Vila and Köpf relies on the timing difference caused by the event loop of the render process, thus only tabs or windows sharing the same rendering process can be attacked. In contrast, our interrupt-timing attack is not restricted to the browser and its child processes as it allows monitoring every other event triggering interrupts on the target device. Moreover, our attack also provides a much higher resolution, which allows detecting interrupts triggered by user input more reliably.

Figure 8.8 shows a trace of a victim opening a website running the measurement code in Chrome on the Xiaomi Redmi Note 3. In addition, the victim opens a tab in incognito mode and taps the screen multiple times. We can even detect these user interactions in different tabs as the attack takes advantage of web workers which are not throttled when running in the background. Thus, the incognito mode offers no protection against our attack.

In the next scenario, we show that our attack is not restricted to processes of the browser application but can be used to spy on every other application as well. The victim visits the website running the measuring application in the Firefox app on the Xiaomi Redmi Note 3 and continues using the phone, switching to other tabs or applications, and later locks the screen. After some time the victim turns on the screen again, where the lock screen prompts the victim for the PIN code. Finally, the victim enters the PIN code, unlocking the phone. Figure 8.9 shows a trace of this scenario. We can clearly observe when the screen is turned off as the CPU

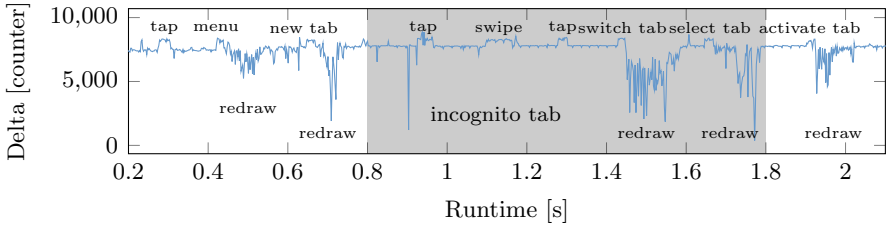


Figure 8.8.: Keystroke timing attack running while switching to a different tab in the Chrome browser on the Xiaomi Redmi Note 3.

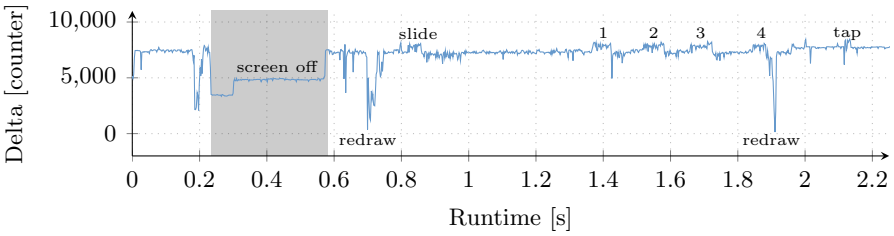


Figure 8.9.: Keystroke timing attack running in the Firefox browser on the Xiaomi Redmi Note 3. While the user locked the screen, the application still detects keystrokes as long as it is executed on the last used tab. The application extracts the exact inter-keystroke timings for the PIN input used to unlock the device.

frequency is lowered to save battery, as well as when the screen is turned on again. Furthermore, we can extract the exact timestamps where the victim entered the 4-digit PIN and the subsequent redraw event.

4.4. Covert Channel

In our fourth experiment, we implement a covert communication channel based on our attack. This allows us to estimate the maximum number of interrupts we can detect. We establish a unidirectional communication with one sender and one receiver. The receiver simply mounts the interrupt-timing attack to sense any interrupts. The sender has to issue interrupts to send a '1'-bit or idle to send a '0'-bit. There is no JavaScript API which allows to explicitly issue interrupts, thus we require an API that implicitly issues an interrupt.

We use `XMLHttpRequest`s to fetch a network resource from an invalid

URL. Every `XMLHttpRequest` which cannot be served from the cache will create a network connection and therefore issue I/O interrupts. Even if the URL cannot be resolved, either because there is no Internet connection, or the URL is invalid, we are able to see the I/O interrupts. Such a covert channel based on hardware interrupts circumvents several protection mechanisms found in modern browsers.

Cross-tab channel. Using the covert channel across tabs breaks two security mechanisms. First, the same origin policy—which prevents any communication between scripts from different domains—does not apply anymore. Thus, scripts can communication across domain borders. Second, due to the security model of browsers, there is no way a HTTPS page is able to load HTTP content. For the covert channel, this security model does not hold anymore.

Cross-browser channel. As the interrupt-timing is not limited to a process, the covert channel circumvents policies such as proccess-per-site or process-per-tab which prevent sites or tabs from sharing process resources. The covert channel can even be used as a cross-browser communication channel. We tested a transmission from Firefox to Chrome and achieved the same transmission rate as in the cross-tab scenario. The communication channel can also be established with a browser instance running in incognito mode.

In all scenarios, the receiver uses a constant sampling interval of 40 ms per bit, resulting in a raw transmission rate of 25 bps. Thus, we are also able to spy on 25 interrupts per second in all those scenarios which is sufficient to monitor keystrokes of even the fastest typists [33]. To reliably transmit data over the covert channel, we can apply the techniques proposed by Maurice et al. [26].

5. Countermeasures

5.1. A Fine-grained Permission Model for JavaScript

In order to impede and mitigate our interrupt-timing attack and other similar side-channel attacks in JavaScript, we propose a more fine-grained permission model for JavaScript running in web browsers. For instance, the existing permission system of Firefox only allows managing the access

control to a limited number of APIs. However, as many websites do not require functionality such as web workers. The user should be capable to allow on a per-page level such features. If an online advertisement running potential malicious code requests for permissions to uncommon APIs, the fine-grained permission system prevents its further execution.

5.2. Generic Countermeasures

Myers [29] evaluated how various user-mode keylogging techniques in malware on Windows are implemented and suggested to generate random keyboard activity by injecting phantom keystrokes that will be intercepted by the malware. Furthermore, Ortolani [31] analyzed the statistical properties of noise necessary to impede the detection of real keystrokes in a noisy channel. While both do not protect against the interrupt-timing attack, Schwarz et al. [36] published a proof-of-concept countermeasure that aims to protect against this type of attacks. The countermeasure injects a large number of fake keystrokes that propagate through the kernel driver up to the user space application. We have verified that the countermeasure successfully injects fake keystrokes that cannot be distinguished from real interrupts by our implementation. Figure 8.10 shows a trace measured on the Google Nexus 5 with the countermeasure enabled. While this countermeasure appears to prevent this attack on personal computers as well, it remains unclear whether it closes the side channel on the Xiaomi Redmi Note 3 where the CPU gets overclocked for every touchscreen input. As the implementation of the countermeasure only supports the touchscreen of the Google Nexus 5 and the OnePlus 3T, we could not evaluate it against our attack on the Xiaomi Redmi Note 3. Therefore, we were unable to verify whether the fake keystrokes injected by the countermeasure also trigger the CPU overclocking and, thus, if the countermeasure protects against this attack on devices with such a behavior.

Kohlbrenner and Shacham [24] implemented the fuzzy time concept [18, 42] in order to eliminate high-resolution timers. While this would prevent our attack in its current implementation, we could use the experimental `SharedArrayBuffers` as suggested by Schwarz et al. [37] and Gras et al. [13] in order to obtain a resolution of up to 2 ns and, thus, to re-enable our attack.

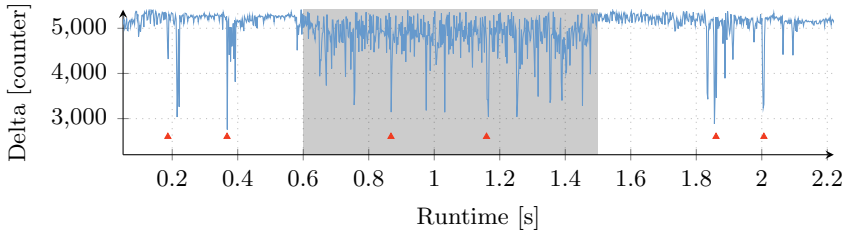


Figure 8.10.: Measurement of the keystroke timing attack running in the Chrome Browser on the Google Nexus 5. The red rectangles show when the user tapped the screen. In the gray area, we enabled the countermeasure [36], making it infeasible to distinguish real keystrokes from fake keystrokes.

6. Conclusion

In this paper, we presented the first JavaScript-based keystroke timing attack which is independent of the browser and the operating system. Our attack is based on capturing interrupt timings and can be mounted on desktop machines, laptops as well as on smartphones. Because of its low code size of less than 256 bytes, it can be easily hidden within modern JavaScript frameworks or within an online advertisement, remaining undetected by the victim. We demonstrated the potential of this attack by inferring accurate timestamps of keystrokes as well as taps and swipes on mobile devices. Based on these keystroke traces, we built classifiers to detect which websites a user has visited and to identify different users time-sharing a computer. Our attack is highly practical, as it works while the browser is running in the background, allowing to spy on other tabs and applications. As the attack is also executed when the phone is locked, we demonstrated that we can monitor the PIN entry that is used to unlock the phone. Finally, as a solution against our attack and other similar side-channel attacks in JavaScript, we proposed a fine-grained permission model for browsers.

Acknowledgments

We would like to thank our anonymous reviewers for their valuable feedback. This project has been supported by the COMET K-Project DeSSnet (grant No 862235) conducted by the Austrian Research Promotion

Agency (FFG) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

References

[1] Alex Christensen. *Reduce resolution of performance.now.* 2015. URL: https://bugs.webkit.org/show_bug.cgi?id=146531.

[2] Alexa Internet, Inc. *The top 500 sites on the web.* Dec. 2016. URL: http://www.alexa.com/topsites.

[3] Kamran Ali, Alex X. Liu, Wei Wang, and Muhammad Shahzad. "Keystroke Recognition Using WiFi Signals". In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking.* MobiCom'15. 2015.

[4] N. S. Altman. "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression". In: *The American Statistician* 46.3 (1992), pp. 175–185. DOI: 10.1080/00031305.1992.10475879.

[5] Donald J. Berndt and James Clifford. "Using Dynamic Time Warping to Find Patterns in Time Series". In: *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining.* 1994.

[6] Jo Malcolm Booth. *Not So Incognito: Exploiting Resource-Based Side Channels in JavaScript Engines.* Bachelor Thesis, Harvard School of Engineering and Applied Sciences. 2015.

[7] Boris Zbarsky. *Reduce resolution of performance.now.* 2015. URL: https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab.

[8] Andrew Bortz and Dan Boneh. "Exposing private information by timing web applications". In: *WWW'07.* 2007.

[9] Wendy Chen and Weide Chang. "Applying hidden Markov models to keystroke pattern analysis for password verification". In: *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration.* 2004.

[10] Chromium. *window.performance.now does not support sub-millisecond precision on Windows.* 2015. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110.

[11] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. "No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis". In: *S&P'16.* 2016.

[12] Edward W Felten and Michael A Schneider. “Timing attacks on web privacy”. In: *CCS’00*. 2000.

[13] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. “ASLR on the Line: Practical Cache Attacks on the MMU”. In: *NDSS’17*. 2017.

[14] Daniel Gruss, David Bidner, and Stefan Mangard. “Practical Memory Deduplication Attacks in Sandboxed JavaScript”. In: *ESORICS’15*. 2015.

[15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.

[16] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. “Scriptless attacks: stealing the pie without touching the sill”. In: *CCS’12*. 2012.

[17] Michael Augustus Hogye, Christopher Taddeus Hughes, Joshua Michael Sarfaty, and Joseph David Wolf. *Analysis of the Feasibility of Keystroke Timing Attacks over SSH Connections*. Tech. rep. School of Engineering and Applied Science University of Virginia, 2001.

[18] Wei-Ming Hu. “Reducing timing channels with fuzzy time”. In: *Journal of Computer Security* (1992).

[19] Syed Idrus, Estelle Cherrier, Christophe Rosenberger, and Patrick Bours. “Soft Biometrics for Keystroke Dynamics: Profiling Individuals While Typing Passwords”. In: *Computers & Security* 45 (2014), pp. 147–155. ISSN: 0167-4048.

[20] Suman Jana and Vitaly Shmatikov. “Memento: Learning Secrets from Process Footprints”. In: *S&P’12*. 2012.

[21] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. “An empirical study of privacy-violating information flows in JavaScript web applications”. In: *CCS’10*. 2010.

[22] Yaoqi Jia, Xinshu Dong, Zhenkai Liang, and Prateek Saxena. “I know where you’ve been: Geo-inference attacks via the browser cache”. In: *IEEE Internet Computing* 19.1 (2015), pp. 44–53.

[23] Pawel Kobojek and Khalid Saeed. “Application of Recurrent Neural Networks for User Verification based on Keystroke Dynamics”. In: *Journal of Telecommunications and Information Technology* 3 (2016), p. 80.

[24] David Kohlbrenner and Hovav Shacham. “Trusted Browsers for Uncertain Times”. In: *USENIX Security Symposium*. 2016.

[25] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security Symposium*. 2016.

[26] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: *NDSS'17*. 2017.

[27] Maryam Mehrnezhad, Ehsan Toreini, Siamak F Shahandashti, and Feng Hao. "Touchsignatures: identification of user touch actions and pins based on mobile sensor data via javascript". In: *Journal of Information Security and Applications* (2016).

[28] Mike Perry. *Bug 1517: Reduce precision of time for Javascript.* 2015. URL: `https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517`.

[29] Mike Myers. "Anti-Keylogging with Random Noise". In: *PoC||GTFO*. Vol. 0x14. 2017.

[30] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications". In: *CCS'15*. 2015.

[31] Stefan Ortolani. "NoisyKey: Tolerating Keyloggers via Keystrokes Hiding". In: *USENIX Workshop on Hot Topics in Security – HotSec*. 2012.

[32] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *USENIX Security Symposium*. 2016.

[33] Svetlana Pinet, Johannes C. Ziegler, and F.-Xavier Alario. "Typing Is Writing: Linguistic Properties Modulate Typing Execution". In: *Psychon Bull Rev* 23.6 (Apr. 2016), pp. 1898–1906.

[34] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds". In: *CCS'09*. 2009.

[35] David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, eds. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, 1986. ISBN: 0-262-68053-X.

[36] Michael Schwarz, Moritz Lipp, Gruss Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. *KeyDrown: Eliminating Keystroke Timing Side-Channel Attacks*. 2017.

[37] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *FC'17*. 2017.

[38] Laurent Simon, Wenduan Xu, and Ross Anderson. "Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards". In: *Proceedings on Privacy Enhancing Technologies* (2016).

[39] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. "Timing Analysis of Keystrokes and Timing Attacks on SSH". In: *USENIX Security Symposium*. 2001.

[40] Paul Stone. "Pixel perfect timing attacks with HTML5". In: *Context Information Security (White Paper)* (2013).

[41] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. "The clock is still ticking: Timing attacks in the modern web". In: *CCS'15*. 2015.

[42] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. "Eliminating fine grained timers in Xen". In: *CCSW'11*. 2011.

[43] Pepe Vila and Boris Köpf. "Loophole: Timing Attacks on Shared Event Loops in Chrome". In: *USENIX Security Symposium*. 2017.

[44] W3C. *High Resolution Time Level 2*. 2016. URL: `https://www.w3.org/TR/hr-time/`.

[45] W3C. *Web Workers - W3C Working Draft 24 September 2015*. 2015. URL: `https://www.w3.org/TR/workers/`.

[46] Zachary Weinberg, Eric Y Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. "I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks". In: *S&P'11*. 2011.

[47] John C Wray. "An analysis of covert timing channels". In: *Journal of Computer Security* 1.3-4 (1992), pp. 219–232.

[48] Xiaopeng Xi, Eamonn Keogh, Christian Shelton, Li Wei, and Chotirat Ann Ratanamahatana. "Fast Time Series Classification Using Numerosity Reduction". In: *Proceedings of the 23rd International Conference on Machine Learning*. 2006.

[49] Kehuan Zhang and XiaoFeng Wang. "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems". In: *USENIX Security Symposium*. 2009.

9

PLATYPUS Software-based Power Side-Channel Attacks on x86

Publication Data

Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. "PLATYPUS: Software-based Power Side-Channel Attacks on x86". In: *IEEE S&P*. 2021

Contributions

Main author.

PLATYPUS: Exploiting Software-based Power Side Channels on x86

[1]Moritz Lipp, [1]Andreas Kogler, [2]David Oswald, [1]Michael Schwarz, [1]Catherine Easdon, [1]Claudio Canella, [1]Daniel Gruss

[1] Graz University of Technology [2] The University of Birmingham, UK

Abstract

Power side-channel attacks exploit variations in power consumption to extract secrets from a device, e.g., cryptographic keys. Prior attacks typically required physical access to the target device and specialized equipment such as probes and a high-resolution oscilloscope.

In this paper, we present PLATYPUS attacks, which are novel software-based power side-channel attacks on Intel server, desktop, and laptop CPUs. We exploit unprivileged access to the Intel Running Average Power Limit (RAPL) interface that exposes values directly correlated with power consumption, forming a low-resolution side channel.

We show that with sufficient statistical evaluation, we can observe variations in power consumption, which distinguish different instructions and different Hamming weights of operands and memory loads. This enables us to not only monitor the control flow of applications but also to infer data and extract cryptographic keys. We demonstrate how an unprivileged attacker can leak AES-NI keys from Intel SGX and the Linux kernel, break kernel address-space layout randomization (KASLR), infer secret instruction streams, and establish a timing-independent covert channel. We also present a privileged attack on mbed TLS, utilizing precise execution control to recover RSA keys from an SGX enclave. We discuss countermeasures and show that mitigating these attacks in a privileged context is not trivial.

1. Introduction

The concept of extracting data from a computer system by monitoring side-channel information, such as its power consumption or electromagnetic emissions, is known since World War II [1]. Power analysis attacks were first presented in an academic context by Kocher et al. [49]

for attacks on cryptographic implementations in smart cards. Subsequent research applied these attacks to different devices and algorithms, particularly to supposedly side-channel-resistant encryption-scheme implementations [23, 25]. However, until recently, power analysis attacks had two limitations. First, they primarily targeted small embedded microcontrollers rather than more complex high-performance desktop and server CPUs. Second, software-based attacks relying on the available interfaces [55, 68, 87] were so far not successfully applied on x86 to leak fine-grained information, e.g., cryptographic key bits.

Software-based power side-channel attacks have been demonstrated on mobile devices for website [68] and app fingerprinting [87], UI inference [87], password length guessing [87], and geolocation estimation [87]. More recently, O'Flynn [63] recovered secrets processed in the secure world on an ARM TrustZone-M platform using an onboard ADC, and Mantel et al. [55] distinguished different RSA keys by measuring the power consumption on Intel desktop machines. The experimental results of Mantel et al. on RSA demonstrated that certain multiply operations of the square-and-multiply implementation can be detected, but no full key recovery was achieved. Similarly, Fusi [19] tried to recover RSA-16384 keys but concluded that the sampling rate of the interface is too low to mount an attack.

In this work, we present PLATYPUS[1] attacks which are novel software-based power side-channel attacks on Intel servers, desktops, and laptops by abusing unprivileged access to Intel's RAPL interface. By observing changes in power consumption with a resolution of up to 20 kHz, we show that different executed instructions and features of their operands can be distinguished. Furthermore, we observe that when a register is filled with data from a cache line, the Hamming weight, *i.e.*, the number of bits set to one, of the loaded value measurably influences the power consumption. We show how these power differences between different operands and load values enable the inference of inputs and intermediate values used for multiplications or masks in an encryption algorithm. We present the building blocks to enable the creation of power traces at instruction-level granularity and develop novel techniques for RAPL power analysis attacks on enclaved and non-enclaved execution.

To demonstrate the applicability of these attacks, we successfully recover AES-NI keys from an SGX enclave and the Linux kernel in 26 hours. In

[1] Power Leakage Attacks: Targeting Your Protected User Secrets

a privileged attack context, we recover RSA private keys from mbed TLS within 100 minutes by inferring the instructions executed inside SGX from a power trace with instruction-level granularity. We derandomize the kernel address space within 20 seconds by observing that accesses to valid and invalid kernel addresses from user space expose a different power consumption footprint. Furthermore, we demonstrate that RAPL enables victims to be observed at sub-cache-line granularity and use this to establish a timing-independent covert channel with a transmission rate of 18.7 bit/s. While an unprivileged attack can be prevented by restricting access to the interface, mitigating privileged attacks is not trivial. We discuss different countermeasures and mitigation strategies for the presented attacks.

To summarize, we make the following contributions:

1. We improve software-based power side-channel attacks to distinguish instructions, operands, and data.

2. We show that the RAPL interface provides sufficient resolution for practical attacks on Intel CPUs.

3. We demonstrate an attack on a cryptographic implementation running in Intel SGX, recovering RSA private keys from mbed TLS within 100 minutes.

4. We show that an unprivileged attacker can use Correlation Power Analysis to recover keys from an AES-NI implementation in an SGX enclave and the Linux kernel within 26 hours (when minimal I/O noise is present) to 277 hours (under real-world conditions).

5. We break kernel address space layout randomization (KASLR) from user space within 20 seconds, observe intra-cache-line accesses, and demonstrate a timing-independent covert channel.

Responsible Disclosure. We responsibly disclosed our findings to Intel on November 16th, 2019. Intel acknowledged our findings and verified our experiments. The issues are tracked under CVE-2020-8694 and CVE-2020-8695 and were held under embargo until November 10th, 2020. We responsibly disclosed our findings to AMD on June 6th, 2020, who tracked this issue under CVE-2020-12912.

Outline. Section 2 provides background. Section 3 analyzes the information leakage induced by the Intel RAPL interface. Section 4 presents the threat model, attack overview, and building blocks. Section 5 evaluates these building blocks and constructs concrete attacks with them. Countermeasures and related work are discussed in Section 6 and Section 7, respectively. We conclude in Section 8.

2. Background

In this section, we provide background on power analysis, Intel RAPL, and Intel SGX.

2.1. Power Analysis

Power analysis attacks are built upon the observation that the power consumption of CMOS digital circuits is data-dependent *by design*. Each bit flip requires one or more voltage transitions from 0 to high (or vice versa). Different data values typically entail differing numbers of bit flips and therefore produce distinct power traces. Equation (9.1) presents the primary sources of power consumption, where α is the probability of a voltage transition, C is the load capacitance, V_{dd} is the supply voltage, F is the clock frequency, I_{sc} is the short-circuit current (when NMOS and PMOS transistors are active simultaneously) and I_{leak} is the leakage current [13].

$$\begin{aligned} P &= (P_{switching}) + (P_{short-circuit} + P_{leakage}) \\ &= \alpha \cdot C \cdot V_{dd}^2 \cdot F + I_{sc} \cdot V_{dd} + I_{leak} \cdot V_{dd} \end{aligned} \tag{9.1}$$

Crucially, $P_{switching}$ with its data-dependent α value is significantly larger than the other terms. Therefore, any circuit not explicitly designed to be resistant to power attacks has data-dependent power consumption. However, in a complex circuit, the differences can be so slight that they are difficult to distinguish from a single trace, particularly if an attacker's sampling rate is limited. Therefore, it is necessary to use statistical techniques such as Differential Power Analysis and Correlation Power Analysis across multiple power traces.

Simple Power Analysis (SPA). In SPA attacks [49], secret-dependent power consumption differences during an operation, e.g., a cryptographic

signature computation, are directly analyzed from power traces to determine the underlying secret. For example, there may be a detectable spike in power consumption when the key bit multiplied is 1 versus when it is 0 because the implementation executes a different instruction sequence in each case. Using SPA, the secret can be extracted with only a small number of traces. However, this is only possible if the secret has a significant impact on the power consumption of the device, and the traces are relatively noise-free. Noise can be averaged out by aligning the traces and computing the mean of the collected traces.

Differential Power Analysis (DPA) and Correlation Power Analysis (CPA). DPA attacks [49] are based on a statistical analysis of a large number of traces with varying input data. Rather than analyzing individual power traces along the time axis as in a typical SPA attack, DPA analyzes how the power consumption at fixed moments in time is a function of the secret data being processed [54]. DPA is significantly more powerful than SPA, as small secret-dependent biases can be detected even in the presence of noise. In our measurement context for power attacks against the CPU, this is relevant for the analysis of operand-dependent power consumption, as these differences are much smaller than the power differences between instructions and can be hidden by measurement error and noise. However, using DPA, these differences can still be identified and used to recover the underlying secret data. CPA [10] is an extension of DPA, which examines the correlation between variations in the set of traces and a leakage model depending on the value of intermediate values [48]. We further explain the inner workings of CPA in Section 5.2.

2.2. Intel RAPL

The Intel Running Average Power Limit (RAPL) mechanism was introduced with the Sandy Bridge microarchitecture to ensure the CPU remains within desired thermal and power constraints [26]. Since Haswell, it has provided three distinct capabilities for controlling average power over timescales of multiple seconds, ~10 ms, and <10 ms (PL1, PL2, and PL3, respectively). These three control loops dynamically adjust the CPU frequency to maximize performance while ensuring the running power average is within each of their (configurable) limits. By design, this modifies the voltage and power consumption. To implement these control loops, it is necessary to provide power-measurement feedback [26]. This can be

done with an analog circuit, e.g., voltage regulator current monitoring, or estimating the energy consumption in the core, as done in Sandy Bridge and Ivy Bridge [26].

Intel defines four different domains for RAPL [39]: package (*PKG*), power planes (*PP0* and *PP1*), and *DRAM*. The package domain estimates energy consumption for the entire socket. PP0 contains the energy consumption estimates of the cores while, on client systems, the PP1 domain refers to a specific device's power plane in the uncore. In this work, PP0 is subsequently referred to as the core domain. On Skylake, Intel has introduced the `PSys` domain covering the entire SoC.

Intel CPUs also provide other functionality for dynamic frequency and voltage scaling (DFVS). For example, they support configurable processor performance states (P-states), as defined in the Advanced Configuration and Power Interface (ACPI) specification [78]. Each state specifies a frequency and voltage operating point [15]. When enabled, the Intel Turbo Boost feature adjusts each core's P-state automatically.

2.3. Intel SGX

Intel SGX (Software Guard Extensions) is an instruction set extension that provides a mechanism for confidentially executing code on a system, isolated from other software on the CPU [39]. The SGX threat model assumes that even privileged software such as the operating system, administrative users, and peripheral hardware may be compromised and behave maliciously. An application using SGX is split into two distinct parts, an untrusted part (which launches enclaves as needed to process secrets) and a trusted part (within an enclave). Each enclave operates within an encrypted and isolated memory region to protect application secrets from hardware attackers. As neither the operating system nor any other application is trusted under the SGX threat model, the processor guarantees that the enclave's memory cannot be accessed by anything but the enclave itself. Additionally, encryption ensures that enclave memory cannot be read directly from the DRAM module, as even peripheral hardware may be malicious. Intel generally considers physical side-channel attacks on SGX out of scope. Side channels [8, 73], race conditions [72, 85], and memory-safety violations [50] are not in the threat model, and it is the developer's responsibility to defend against these.

Table 9.1.: RAPL register update intervals if accessed directly in the kernel or via the *powercap* driver.

Register	Measurement Unit	Kernel	Driver
MSR_PKG_ENERGY_STATUS	μJ	1000 µs	1000 µs
MSR_DRAM_ENERGY_STATUS	μJ	1000 µs	1000 µs
MSR_PP0_ENERGY_STATUS	μJ	50 µs	50 µs
MSR_PERF_STATUS (core voltage)	V	150 µs	-

3. Intel RAPL Leakage Analysis

In this section, we analyze the power side-channel information leakage from Intel RAPL data, considering both user-space and SGX-enclave targets. We experimentally evaluate that we can distinguish and fingerprint both individual instructions (Section 3.3) and the influence of their operand values (Section 3.4). Furthermore, we evaluate the influence of concrete data values on energy consumption (Section 3.5) as well as the influence of the cache status of a memory address in a load operation (Section 3.6).

While energy-consumption interfaces also exist on non-Intel CPUs, we focus on Intel's RAPL implementation and briefly discuss other architectures in Section 7.2.

Note that while we primarily refer to runtime *energy* consumption rather than *power* consumption throughout this work, these are directly related, as $power = energy \div time$.

3.1. RAPL Interface

RAPL provides an interface both for controlling the core frequency and voltage and for monitoring the power consumption of the socket and memory domain (see Section 2.2). To date, Intel RAPL has typically been used to model energy consumption on a system level [67] or in benchmarks [46].

We can read the RAPL register values to measure energy consumption, *i.e.*, the cumulative power consumption over a sampling period, in two ways:

- **Unprivileged Access**: On Linux, the power capping framework `powercap` provides unprivileged access to Intel RAPL by exposing the MSRs through the `sysfs` interface. This allows an unprivileged attacker

to directly read the value of the individual packages from a file located in the `/sys/devices/virtual/powercap` tree.

- **Privileged Access**: A privileged attacker targeting Intel SGX can load a kernel module to read the RAPL MSRs.

While measuring the update intervals of the values provided by both the Linux RAPL user-space driver and by accessing the MSRs directly, we observed that several values update faster than the documented RAPL update rate of 1 ms. We observe that the `MSR_PP0_ENERGY_STATUS` (core energy consumption) and `MSR_PERF_STATUS` (core voltage) values update substantially faster, at 50 µs and 150 µs intervals, respectively. The results of this evaluation are shown in Table 9.1. These rates were consistent across the different tested microarchitectures.

3.2. Experimental Setup

Throughout this work, we tested on Intel mobile, desktop, and server CPUs. Table 9.2 provides details of each Intel CPU used in our experiments. In the mobile setting, we tested on a Lenovo Thinkpad T480s and T495s, both using Core i7-8650U CPUs, on a Lenovo Thinkpad T460s with a Core i7-6600U and an Intel NUC7I3BNH using a Core i3-7100U. For the desktop setting, we evaluated a system using a Core i5-3230M, a Core i7-6700K, and a Core i9-9900K. Finally, for the cloud server setting, we evaluated 3 systems, with Xeon E3-1240 v5, Xeon E3-1275 v5, and Xeon Silver 4214 CPUs. All tested devices run Ubuntu Linux, with versions from Ubuntu 16.04 to Ubuntu 20.04, and kernels 4.15.0 to 5.4.0. Different Ubuntu versions and kernels did not appear to influence the results, and we would only expect this to occur if there were a substantial update to the behavior of the `powercap` driver.

Unless stated otherwise, all systems were using the default system configuration, and all mobile systems were connected to an AC power source. For example, we did not fix the CPU frequency or disable Intel Turbo Boost.

3.3. Distinguishing Instructions

With our first experiment, we demonstrate that Intel's RAPL interface enables distinguishing different instructions via their energy consumption. To measure the energy consumption of an instruction, we record

Table 9.2.: CPU type, model, and microarchitecture for each device under test, and whether it leaks data of operands, type of instructions, and the target of memory loads.

Type	CPU	Microarchitecture	Leakage		
			Data	Instr.	Target
Mobile	Core i7-6600U	Skylake	✓	✓	✓
Mobile	Core i3-7100U	Kaby Lake	✓	✓	✓
Mobile	Core i7-8650U	Kaby Lake-R	✓	✓	✓
Desktop	Core i5-3230M	Ivy Bridge	✗	✓	✓
Desktop	Core i7-6700K	Skylake-S	✓	✓	✓
Desktop	Core i9-9990K	Coffee Lake-R	✓	✓	✓
Server	Xeon E3-1240 v5	Skylake	✓	✓	✓
Server	Xeon E3-1275 v5	Skylake	✓	✓	✓
Server	Xeon Silver 4214	Cascade Lake	✓	✓	✓

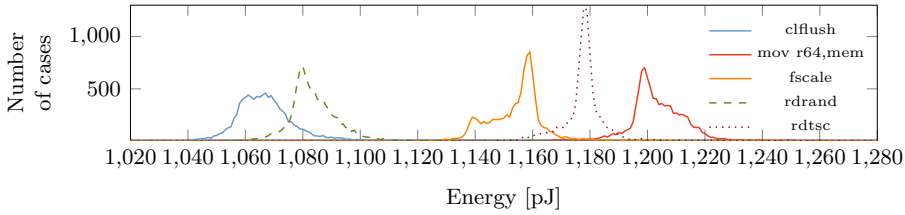


Figure 9.1.: A histogram of the power consumption of various instructions on the i7-6700K (desktop) system.

its energy consumption over 10 000 consecutive executions and take the median value to eliminate system-level noise, e.g., erroneous high values caused by interrupt handling or the process being descheduled. We observe the energy consumption across the entire CPU package to ensure that non-core activity, e.g., DRAM access, is included.

Table 9.3 lists the measured energy consumption of different instructions on our i7-6700K (desktop), Xeon Silver 4214 (server), and i7-8650U (mobile) systems. We can clearly observe inter-instruction differences in energy consumption. This enables an attacker to identify which instructions are executed, provided they can profile the energy consumption of the victim microarchitecture. For instance, the `rdtsc` instruction consumes 0.1189 nJ on the i7-6700K, versus 0.1864 nJ on the Xeon Silver 4214 and 0.0848 nJ on the i7-8650U. As illustrated in Figure 9.1, this clearly distinguishes it from `rdrand` and `clflush`, which have much lower average energy consumption. However, as some instructions have similar energy

Table 9.3.: Average observed energy consumption (package domain) of different instructions on our i7-6700K (desktop), Xeon Silver 4214 (server), and i7-8650U (mobile) systems.

Instruction	Xeon Silver 4214	i7-6700K	i7-8650U
`nop`	0.1795 nJ	0.1189 nJ	0.0843 nJ
`inc r64`	0.1795 nJ	0.1208 nJ	0.0858 nJ
`xor r64, r64`	0.1795 nJ	0.1209 nJ	0.0849 nJ
`mov r64, mem`	0.1868 nJ	0.1247 nJ	0.0840 nJ
`imul r64, r64`	0.1798 nJ	0.1169 nJ	0.0887 nJ
`fscale`	0.1867 nJ	0.1182 nJ	0.0877 nJ
`rdrand r64`	0.1797 nJ	0.1129 nJ	0.0982 nJ
`rdtsc`	0.1864 nJ	0.1189 nJ	0.0848 nJ
`clflush mem`	0.1865 nJ	0.1129 nJ	0.1018 nJ
`aesenc xmm, xmm`	0.1794 nJ	0.1188 nJ	0.0946 nJ

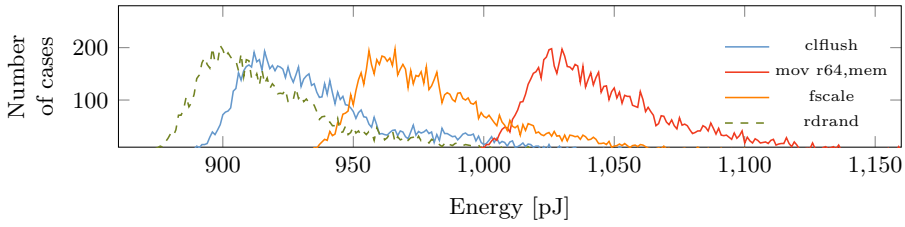


Figure 9.2.: A histogram of the power consumption of various instructions inside an SGX enclave on our i7-8650U (mobile).

consumption, this method may identify multiple instruction candidates. For example, on the Xeon Silver 4214, `nop`, `inc`, and `xor` are indistinguishable at this measurement granularity. While the table only shows the values for when the mobile system (i7-8650U) is connected to an AC power source, we also observed these differences when running on battery power. As not every instruction sequence has the same probability, it may be possible to recover individual instructions using heuristics for typical instruction sequences or by leveraging existing research regarding distinguishing x86 code sequences from data bytes [84].

These results align with those of prior work, in which the different energy consumption of instructions was identified using either Intel RAPL [19, 30, 35, 58] or dedicated hardware [6, 74, 77, 82].

Differing power consumption can also be observed for instructions executed inside SGX enclaves, as shown in Figure 9.2. The enclave's isolation

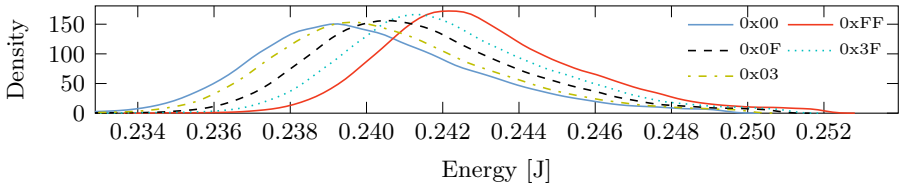


Figure 9.3.: Measured energy consumption of the `imul` instruction with one operand fixed to 8 and the other varying in its Hamming weight.

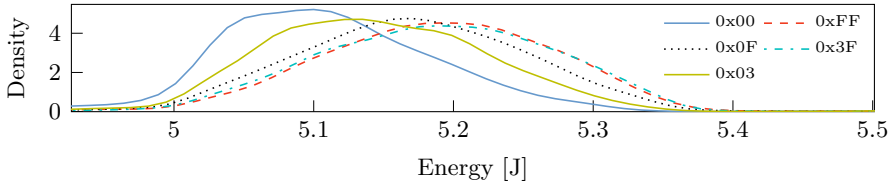


Figure 9.4.: Measured energy consumption of the `shr` instruction with a register set to different Hamming weights.

is no protection here: just like with execution outside the enclave, instructions can be clearly distinguished. Interestingly, energy consumption for the `clflush` instruction is higher inside an SGX enclave, which we attribute to the transparent memory encryption. With other instructions, we do not observe such a difference.

3.4. Distinguishing Operands

In addition to the energy-consumption differences of instructions, the energy consumption of some instructions further depends on their operand value. Intuitively, e.g., integer multiplication should use more energy if more operand bits are set. We measure the `imul` instruction with different operand values in user space on our Xeon E3-1240 v5 system with a fixed core frequency. For the 64-bit operand, we used Hamming weights of 0, 16 (a quarter of the bits), 32 (half of the bits), 48 (three-quarters of all the bits), and 64 (all of the bits). The second operand remains fixed to the value 8. In Figure 9.3, it can be seen that the power consumption differs based on the Hamming weight. While we cannot deduce the exact value of the operand, it reduces the range of potential values, and it can be used in CPA attacks (cf. Section 5.2).

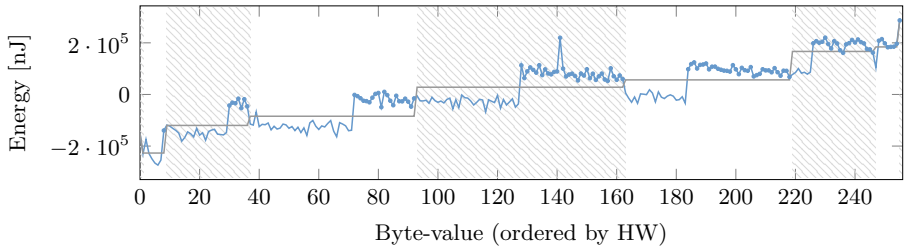


Figure 9.5.: Energy consumption of the `movb` instruction for all byte values, ordered by Hamming Weight (HW) and value. The circle marks values where the most-significant bit is set.

The distinction is not limited to the `imul` instruction. Figure 9.4, for example, shows the differences in power consumption for `shr` on our i7-8650U system with a clear difference in power consumption depending on the Hamming weight of the shifted register. We reproduced these results on an i7-6600U, i7-6700K, and i9-9900K and Xeon 4214 CPU. For the `vpand` instruction, the distributions of the energy consumption differ if one of the operands is zero or not. Ivy Bridge and Sandy Bridge estimate the power consumption [26] and do not rely on hardware probes. Thus, we cannot distinguish operands and data, as we verified on an i5-3230M (cf. Table 9.2).

3.5. Distinguishing Data

We showed that it is possible to fingerprint different instructions and the Hamming weight of their operands. In the third experiment, we evaluate the influence of data values loaded from the cache on the energy consumption. We set up a cache line with alternating 1 and 0 bits to achieve an even Hamming weight. We then set the value of the first byte in the cache line and measure the energy consumption of a memory load of that specific byte, using the `movb` instruction for all 256 value possibilities. To prevent a possible measurement side-effect introduced by the order of the different values measured, we set the value in a pseudo-random order.

We performed the experiment on our Intel Xeon E3-1240 v5 (server) system, collecting measurements for all possible byte values for 627 hours. While the obtained measurements show a trend of increasing energy consumption with increasing value, a power model was not observable. When sorting the values based on their Hamming weight and value, as illustrated

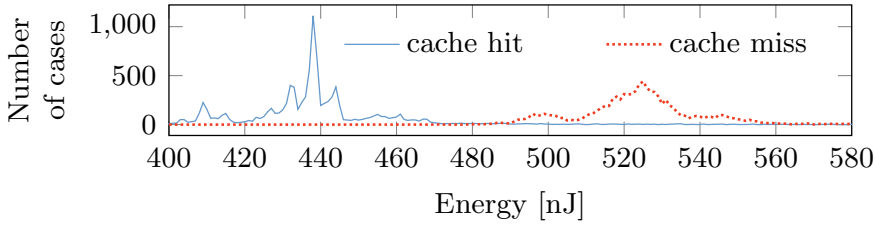


Figure 9.6.: Using RAPL to distinguish whether the target of a memory load is cached (cache hit) or not (DRAM access).

in Figure 9.5, the increasing power consumption is clearly visible. However, one can measure a different power consumption within values of the same Hamming weight (separated in the plot by the white background or gray pattern). These spikes correlate to exactly those values where the most-significant bit is set (data points with circle marks).

To verify the results on other microarchitectures, we performed a reduced experiment with fewer different Hamming Weights (Section 3.4). On the i7-6600U (mobile) system set to a fixed frequency, we observed a similar increasing energy consumption with the Hamming Weight of the byte being read after measuring for 5 minutes.

While we cannot deduce the exact data value that is loaded, one can clearly infer information about the Hamming weight and whether the most-significant bit is set by measuring its energy consumption. Similarly to the varying power consumption we observed with instruction operands. This allows us to constrain the range of potential values.

3.6. Distinguishing Load Targets

To get an even finer granularity when distinguishing instructions, we demonstrate further that it is possible to distinguish the cache status of a load destination. When a memory load accesses data that is already cached, DRAM consumes significantly less energy than when a data access misses the cache and must be first fetched from the main memory.

We evaluated this experiment on several CPUs, as shown in Table 9.2. Figure 9.6 shows a histogram of data fetched from the cache and DRAM on our i7-8650U (mobile) system. When recording power consumption using RAPL on the DRAM domain, there is a clear difference in power consumption for cache hits and cache misses, both when connected to

a power supply and when running on a battery. Hence, code sequences which are vulnerable to cache attacks can also be exploited using power measurements. This allows an attacker to build a timer-free cache attack, similar to the timer-free attacks presented by Diesselkoen et al. [17] and Gruss et al. [27].

4. Attack Overview & Building Blocks

In this section, we introduce the basic concept of PLATYPUS attacks based on the observations from Section 3. We describe the necessary building blocks and their applicability in various scenarios and attacker models before demonstrating several attacks in Section 5.

4.1. Attack Scenarios & Attacker Model

We consider two different attacker models for our attacks: an unprivileged user-space attacker and a privileged kernel-space attacker. For all our attacks, we assume native code execution on an Intel CPU and no software bugs or hardware vulnerabilities.

Unprivileged User-space Attacker. A user-space attacker can run native *unprivileged* code. Hence, the user-space attacker only has access to power interfaces provided by kernel drivers, e.g., the RAPL `sysfs` interface from `powercap`. In addition, the user-space attacker can communicate with other interfaces, e.g., `ioctl`, to the kernel, and interfaces exposed by other applications, e.g., sockets. Furthermore, the user-space attacker could, to some extent, influence other running applications, e.g., by attempting to slow down another process by exhausting its resources [2].

Privileged Kernel-space Attacker. The kernel-space attacker can execute native *privileged* code. Hence, the kernel space has direct access to Intel RAPL's MSRs. The privileged kernel-space attacker has full control over the operating system and, thus, direct access to the memory of running applications. Therefore, we assume an attack on SGX enclaves (see Section 2.3) where the memory is encrypted and cannot be inspected by the operating system. For the SGX enclave, a malicious operating system is in the threat model [16].

4.2. Building Blocks

In this section, we describe the necessary building blocks. We describe how a privileged attacker can achieve precise execution control, enabling them to overcome the low sampling rate faced by an unprivileged attacker. We characterize the documented power interfaces we use for our attacks.

4.2.1. Power Information

To mount PLATYPUS attacks, it is necessary to obtain a power consumption measurement within the software. While throughout this work, we focus on Intel RAPL, these attacks are, in general, not restricted to the Intel platform. We discuss other microarchitectures and interfaces in Section 7.2.

One inherent challenge of software-based power analysis is the low update rate of power data sources in contrast to the frequency of the execution stream under attack (see Section 5). When attempting to reconstruct a signal, it is crucial to sample at a sufficiently high rate. While measuring the `PP0` MSR directly from kernel space, the sample rate is a bit higher; it is still suboptimal. For other attacks, the relevant values are from other domains, e.g., `PKG` and `DRAM`, which do update at the documented slower rate (e.g., Section 3.3).

In general, undersampling means that we cannot obtain samples at a sufficient number of points over the time axis, e.g., because the time axis is very short when sampling only for a few nanoseconds. However, if the attacker can conduct repeated attacks, then multiple traces can be combined to recover an averaged but more complete trace.

Moreover, note that Intel RAPL does not provide the energy consumption per core but per processor package. Thus, code executed on other cores have a direct influence on the measurement of a specific piece of code running on one core, and, thus, the number of overall measurements increases to average out the noise introduced by the other cores. In the case of a privileged attacker, the noise introduced by other cores can be limited as the attacker can disable them or control what code is executed on which core. In contrast, AMD's implementation of RAPL provides per-core counters (cf. Section 7.2).

Note that while factors such as frequency and P-state do influence the raw energy consumption values measured, it is not necessary to fix them,

as the data-dependent differences which our attacks exploit remain observable.

4.2.2. Alignment and Execution Control

In the attack scenario where the attacker measures power consumption in parallel to the victim's execution, the attacker needs to align the recorded traces. The trace needs to contain a distinctive feature, e.g., a distinct peak in power consumption, so that traces can be shifted into alignment with each other. While a privileged attacker can precisely control the victim's execution and interrupt it at will, an unprivileged attacker cannot. However, if the attacker can control when the execution of the attacked code begins, or use a trigger signal such as a cache-based side channel [72], then the collected traces can be aligned based on that timing information.

Precise execution control is the capability to control the victim's execution at instruction-level granularity. To achieve precise execution control of SGX enclaves, we repurpose previously published techniques for microarchitectural attacks and apply them in our software-based power analysis attack.

Single-Stepping. With SGX-Step, Van Bulck et al. [81] introduced the concept of single-stepping SGX enclaves. They achieve this by configuring the local APIC timer interrupt interval so that the interrupt arrives during execution of the first instruction after `eresume`. This triggers an Asynchronous Enclave Exit and execution of an attacker-controlled interrupt handler, where attack-specific code can be executed. This process can be repeated, resuming the enclave to execute precisely one instruction each time. The SGX-Step framework enables these APIC timer interrupts to be configured from user space, along with user-space modification of page table entries. Single-stepping has since been used in a range of microarchitectural attacks. For example, it was used in the Foreshadow attack [79] to extract key material from SGX enclaves to bypass enclave launch control and to forge local and remote attestation. It was further used with LVI [80] to mount a transient fault attack on AES-NI.

Zero-Stepping. If the local APIC timer is configured such that the interrupt arrives within `eresume`, the enclave instruction pointer will not

advance, and so a single instruction can be repeatedly executed for measurements. Zero-stepping can also be achieved by revoking the execute permissions of an enclave's code pages triggering a page fault on the first instruction after `eresume`. Thus, no enclave instruction is actually executed [81]. MicroScope [75] provides an additional technique to replay an enclave instruction repeatedly using a memory access instruction triggering the page fault handler as a *replay handle*.

Zero-stepping provides us with a powerful attack primitive to measure the power consumption of a single instruction repeatedly. We can advance to the desired instruction using single-stepping as described above and then sample the instruction an arbitrary number of times with zero-stepping. Crucially, it enables us to take this arbitrary number of samples *even if we are only able to trigger a single execution of the algorithm under attack in the enclave*. Taking a large number of samples in this way allows to overcome the limited sampling rate and resolution of RAPL.

5. Evaluation

In this section, we combine our attack primitives to build concrete PLATYPUS attacks. We demonstrate that we can recover an RSA key used inside an SGX enclave using mbed TLS (Section 5.1). We use CPA attacks to extract AES keys from the Linux kernel and from an SGX enclave, both utilizing the AES-NI instruction extension (Section 5.2). Furthermore, we exploit Intel RAPL to observe victims at sub-cache-line granularity (Section 5.3), to derandomize the kernel address space (Section 5.4), and to establish a timing-independent covert channel (Section 5.5).

5.1. RSA Key Recovery

In this attack scenario, we consider a privileged attacker targeting an Intel SGX enclave performing RSA signatures. As the threat model of SGX considers the operating system to be untrusted, the attacker is allowed to load arbitrary kernel modules. We consider two different target implementations. First, we will show a toy example imitating a square-and-always-multiply RSA implementation that allows to visually illustrate the leakage observable through the RAPL domain and the core voltage using precise execution control. Second, we will demonstrate an attack on mbed TLS [4] to extract RSA private keys from the SGX enclave. Further, we will discuss scenarios where the code executed within the enclave

is unknown, as well as scenarios where the implementation of the enclave is known by the attacker, thus enabling the attacker to target specific instructions within the enclave's execution.

Setup. In our experiment, the victim provides an API for signing or decrypting user-provided data inside an SGX enclave, making it secure against direct attacks from the operating system, other enclaves, and user space. For simplification and evaluation purposes, we first imitate a square-and-always-multiply RSA implementation that performs the same instructions with different operands based on the value of the currently processed key bit. In our second scenario, we attack the RSA implementation of mbed TLS inside an enclave.

We use SGX-Step [81] and hook the local APIC interrupt `apic_irq` handler to record the values of the timestamp counter `TSC`, the current energy consumption for the desired domain (`MSR_PP0_ENERGY_STATUS`), and the current P-state and core voltage (`MSR_PERF_STATUS`). Further, we hook the Asynchronous Exit Pointer (AEP) to decide if we want to zero-step the current instruction or advance to the next instruction (single-step), as described in Section 4.2.2.

5.1.1. Toy Example

For our toy implementation the number of instructions executed is independent of the bit processed. The key insight here is that even an implementation with these defensive properties against side-channel attacks can still be successfully attacked via the RAPL power side-channel. Specifically, for a 1 bit, we execute two `vpmuludq` instructions, one for a square operation and one for multiplication. For a 0 bit, we execute a `vpmuldq` instruction for the square operation and an additional one using a dummy output register with no architectural effect.

Evaluation. We evaluated this attack scenario on our Xeon E3-1275 v5 (server) system and the i9-9000K (desktop) system. For each execution run of the victim, we single-step to each instruction and measure it over 188 zero steps, *i.e.*, the number of zero steps that need to be executed such that the RAPL counter is updated. We measured over 96 000 execution runs, yielding an overall attack time of 8.11 h on the E3-1275 v5. The result is illustrated in Figure 9.7. One cannot only clearly see the difference

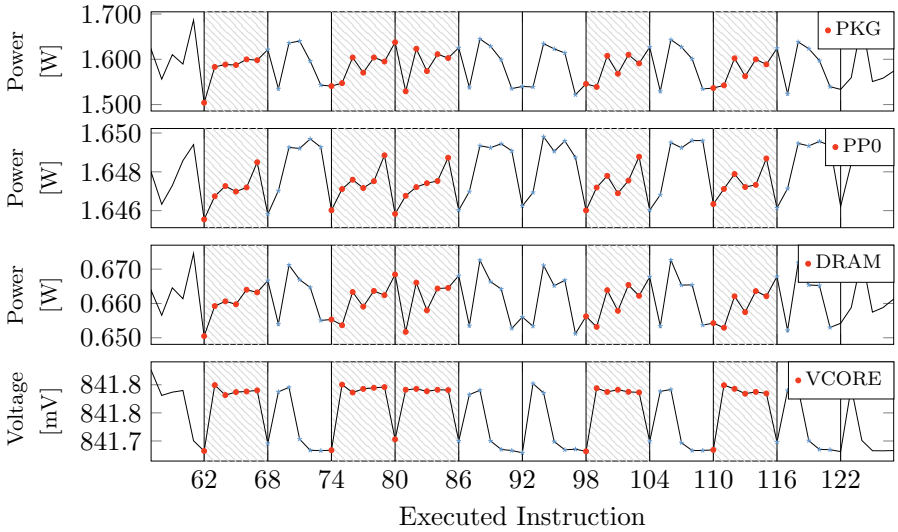


Figure 9.7.: Energy consumption and core voltage per executed instruction of a victim enclave. The attacker uses single-, and zero-stepping to precisely measure single instructions of the victim, allowing to distinguish between them to leak the single key bits. Highlighted areas with red markers indicate measurements for instructions executed for a 1-bit, blue markers indicate a 0-bit.

in power consumption for every instruction measured but also distinguish whether the key bit was set to 1 (highlighted areas with red markers) or 0 (areas with blue markers) by examining the instructions depending on the key bit. This allows recovering the secret key successfully. Furthermore, as shown in Figure 9.7, these differences are not only clearly visible in the different RAPL domains (package, PP0, DRAM) but also in the core voltage.

Under the assumption that the attacker knows which set of instructions needs to be sampled for each key bit, the attacker does not need to zero-step every single instruction. In our example, it would be sufficient just to sample every seventh instruction to recover every single key bit. Even if different instructions are executed depending on the key-bit value, the attacker can advance directly to the instruction responsible for the next key bit after recovering the current key-bit value. To correctly distinguish between these two instructions, we require at least 350 measurements over

255 zero steps when observing the core voltage to recover 99.4 % of the key bits correctly. The number of zero steps is required to obtain an updated power measurement from the RAPL MSRs (see Section 2.2). For the different RAPL domains, we require more traces, e.g., at least 40 000 traces over 188 zero steps to recover 99.5 % of the key bits. Thus, with a runtime of 1.35 ms per trace for each key bit, a 2048-bit RSA attack can be successfully recovered within 16.5 minutes when observing the core voltage. With RAPL and a runtime of 0.99 ms per trace for each key bit, we can successfully recover the key within 23.3 hours. This number highly depends on how many measurements are required to distinguish both cases with a high probability and, thus, can be different in other scenarios.

5.1.2. Attack on mbed TLS

In our second scenario, we extract RSA keys from the mbed TLS [4] (version 2.13.0) implementation with a fixed window length of 1 (`MBEDTLS_MPI_WINDOW_SI 1`). In order to distinguish the key bits, we do not directly target the branch instruction of the fixed-window exponentiation. Instead, we aim at an instruction with a more distinct energy consumption inside the branch. In SGX, Intel's `fast_memset` implementation replaces the standard libc `memset` implementation called inside the `mpi_montmul` function with AVX instructions. AVX instructions are located at a given offset n from the branch instruction if the key bit is set. If the key bit is 0, a different (non-AVX) instruction is executed with the same instruction offset, *i.e.*, the nth instruction following the branch is not an AVX instruction. Thus, we can directly reconstruct the key bit by measuring the energy consumption at the instruction executed with the instruction offset after the branch.

However, the implementation of mbed TLS skips leading zeroes of the exponent and, therefore, has a setup phase depending on the key. Additionally, depending on whether the key bit is 1 or 0, a different number of instructions is executed for each key bit. In order to recover the full private key, we first need to determine the number of zero bits to find the instruction leaking the first key bit correctly. Second, we need to calculate the offset of the next key bit instruction to zero-step based on previously reconstructed key bits.

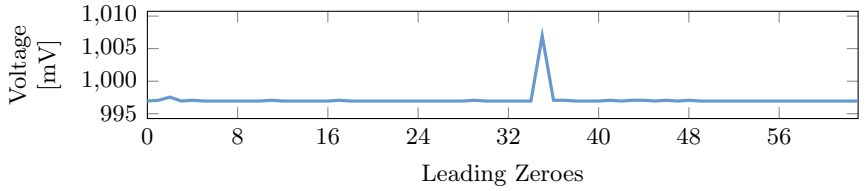


Figure 9.8.: Measured core voltage of all 63 possible leading zero offsets. The spike at offset 35 marks the first set key bit.

Determining the number of zero bits. The mbed TLS implementation skips the leading zeroes of the exponent. Therefore, the offset of the first instruction executed after the key-bit branch depends on the number of leading zeroes. In order to overcome this challenge, we note that the maximum number of leading zeroes relies on the size of the `mbedtls_mpi_uint` data type, which is either 32 or 64 bits. Hence, we assume a possible maximum of 63 leading zeroes. For each possibility, we calculate the offset of the targeted AVX instruction (a 1-bit) under the assumption that we have n leading zero bits. We target each calculated instruction offset and record the core voltage when zero-stepping this instruction. For each measurement, we reset the current energy consumption to a known state by executing multiple `hlt` instructions. Then, we measure each instruction 3 times and use the median of the measured values as a classifier and illustrate the observed measurements in Figure 9.8. The distinct peak, and, thus, the first set key bit gives away the number of leading zeroes.

Offset Oracle. In order to find the instruction to zero-step for the next key bit, we create an oracle that predicts the offset of the next key bit instruction based on previously reconstructed key bits. The oracle receives the known-plaintext input, the public modulus, the number of leading zeroes as well as the current key bit. Using this information, the oracle calculates the next instruction offsets that need to be zero-stepped in order to recover the next key bit.

In our attack, we implemented the oracle utilizing the same enclave implementation for demonstration purposes. We inject the current key hypothesis into the enclave to automatically find the next instruction offset using single stepping. While this increases the runtime of the attack, it allows to predict the next offset without having to analyze the enclave on an instruction-level basis.

Evaluation. We evaluated our attack on a Xeon E3-1275 v5 server CPU. In order to profile the instruction at the calculated offset, we measure the observed core voltage 3 times over 256 zero steps. For a 1-bit, the instruction at and after the given offset are AVX instructions and, thus, we measure both to increase the signal. We used an RSA key pair with a 512-bit modulus for evaluation purposes. In 211 minutes ($n = 5$,$\sigma_{\bar{x}} = 7.2$), we were able to reconstruct the 509 key bits without any error. Figure 9.13 in Section B illustrates one of the recorded traces. Note that the slow implementation of the oracle compensates for 52 minutes ($n = 4$, $\sigma_{\bar{x}} = 6.73$) of the attack. In addition, we successfully recovered the key without any error in 100 minutes, even when we measured each instruction only once.

5.2. Correlation Power Analysis Attacks

The SPA attack in Section 5.1 exploits the comparatively strong change in leakage in the energy consumption or core voltage due to the different instructions executed. In contrast, in this section, we focus on differential attacks (see Section 2.1) that apply to implementations with secret-independent control flow, e.g., symmetric ciphers like AES, targeting the data-dependent leakage of single instructions. We show that Correlation Power Analysis can be applied to exploit the small, data-dependent leakage of single instructions even when capturing one aggregate leakage sample for the whole cryptographic algorithm.

To this end, we demonstrate key recovery attacks against AES-NI, an x86 instruction-set extension designed to mitigate timing and cache side-channel leakage [31] in two different settings. First, we will recover the AES key processed inside an SGX enclave and second, from a Linux kernel module.

In contrast to the RSA signature generation from Section 5.1, a single run of our target algorithms has a very short runtime (on the order of tens to hundreds of cycles). Hence, the overall energy consumption is below the resolution of the RAPL interface (a single invocation usually reads as a zero energy consumption difference). We, therefore, generally measure the aggregate energy consumption of R invocations of our target cipher (typically 16 M) to obtain a single leakage sample p. Our attacks, therefore, apply to situations where an adversary can trigger the encryption/decryption of many blocks of data, e.g., disk and file encryption, encrypted network protocols like TLS, or (un)sealing of large

enclave state. In the case of a privileged attacker, the attacker model allows the alternative approach of using zero-stepping to only repeat the target instruction in the scenario of Intel SGX. Moreover, differential attacks like CPA make use of many leakage samples p_n (traces) for different inputs (plaintexts) x_n for $n < N$. Depending on the scenario, we used N between 2 M and 16 M.

5.2.1. Key extraction with CPA

To recover a secret value, we compute the correlation $\rho(p, h)$ between the observed power consumptions p_n and hypothetical leakage values h_n over all N traces. The choice of h depends on the targeted operation and the leakage characteristics of the target implementation and processor. For example, for recovering byte 0 of the round key in the final round of AES, a common choice (given a key candidate k) is:

$$h_n^k = \mathrm{HW}\left(\mathrm{SBox}^{-1}\left(c_n^0 \oplus k\right)\right) \tag{9.2}$$

where c_n^0 is byte 0 of the n'th ciphertext, and HW denotes the Hamming weight. Computing $\rho^k\left(p, h^k\right)$ for all candidates $k = 0 \ldots 255$, the correct key candidate can be identified as the one with maximum correlation. This process is repeated for each byte. Other choices of h are possible, e.g., when targeting the XOR in the first round of AES:

$$h_n^k = \mathrm{HW}\left(x_n^0 \oplus k\right) \tag{9.3}$$

For a given number of traces N, the *noise level* is [54]:

$$\rho_{noise} = \frac{4}{\sqrt{N}} \tag{9.4}$$

Only correlations $\rho \geq \rho_{noise}$ are considered significant. Assuming an ideal correlation ρ_{exp} that captures only the correlation between the target value and a noise-free trace and a Signal-to-Noise Ratio (SNR), the observed correlation ρ can be computed as:

$$\rho = \frac{\rho_{exp}}{\sqrt{1 + {}^1/_{SNR}}} \tag{9.5}$$

5.2.2. SGX Enclave

In the first setting, we will demonstrate AES-NI key recovery on an SGX enclave.

Setup. We implement an enclave that exposes an `ecall` to encrypt a buffer using an in-enclave secret key. It deploys a full AES implementation from Intel's Integrated Performance Primitives (Intel IPP) [41] `ippsAESEncryptECB` function that uses the AES-NI instruction set. While the SGX scenario enables a privileged attacker (Section 4.1), we assume an unprivileged attacker.

We further considered two scenarios:

1. Minimal I/O noise: The unprivileged attacker records the accumulated power consumption of 16 384 calls to `ippsAESEncryptECB`, each encrypting 16 kB, within a single `ecall`.
2. Real-world conditions: The unprivileged attacker records the accumulated power consumption of 64 `ecall` invocations, each encrypting 4 MB with a single call to `ippsAESEncryptECB`.

Profiling. To better understand the leakage behavior of the AES-NI implementation on the processor under attack, we compute the AES state after every round. Further, we compute the correlation between different power models and our observed traces.

We recorded 2 M traces (thus $\rho_{noise} = 0.0028284$) for scenario 1 in 26 h and 16 M traces (thus $\rho_{noise} = 0.001$) for scenario 2 in 277 h. Table 9.4 shows the Hamming weight's correlations for each round and the Hamming distance between rounds on our Xeon E3-1240 for scenario 1.

As discussed in Section 5.2.1, bold entries highlight significant entries with an exploitable statistical dependency ($\rho \geq \rho_{noise}$). In addition, the Significance Factor (SF) is computed as ρ/ρ_{noise}, *i.e.*, $|\mathrm{SF}| \geq 1$ indicates a significant correlation. For instance, the Hamming weight of the input and output leak, as well as the Hamming weight of the 128-bit state after the initial XOR of round key 0 to the plaintext (correlation $\rho = 0.05032280$). In addition, the Hamming distance between the input and output of each AES round leaks, which is crucial for subsequent key recovery attacks.

Table 9.4.: Profiling correlations after 2 M traces for AES-NI in scenario 1 for the Hamming weight (HW) for each round and Hamming distance (HD) between rounds. Bold entries and a $|\text{SF}| \geq 1$ highlight significant statistical dependencies.

HD	ρ	SF	HW	ρ	SF
00 → 01	0.03675729	**13**	00	0.06885782	**24**
01 → 02	0.02006421	**7.1**	01	0.05032280	**18**
02 → 03	0.03676030	**13**	02	0.00145256	0.51
03 → 04	0.03728021	**13**	03	0.00181104	0.64
04 → 05	0.03754657	**13**	04	0.00188247	0.66
05 → 06	0.03739362	**13**	05	0.00186131	0.66
06 → 07	0.03804800	**13**	06	0.00204561	0.72
07 → 08	0.03790153	**13**	07	0.00151157	0.53
08 → 09	0.03810117	**13**	08	0.00250208	0.88
09 → 10	0.03967649	**14**	09	0.00272294	0.96
10 → 11	0.01820413	**6.4**	10	-0.00045022	-0.16
			11	0.08859152	**31**

For scenario 2, we similarly observed Hamming weight and Hamming distance leakages for the AES rounds, albeit with a lower magnitude of the correlations. For example, for the final round, the correlation is $\rho = 0.00532594$ in scenario 2, compared to $\rho = 0.01820413$ in scenario 1. Therefore, for key recovery in scenario 2, a larger number of traces is required. The respective profiling results are given in Table 9.6 in Section C.

Key Recovery. To recover the key, we build a CPA attack using the Hamming distance between the input and the output of the final round of AES. As observed in the profiling phase, the correlation of the Hamming distance of the final round $10 \rightarrow 11$ yields $\rho = 0.01820413$ in scenario 1. In this case, we successfully recovered all 16 bytes of the final round key using 2 M traces, and hence also the actual AES key due to the reversible key schedule of AES.

In scenario 2, the respective correlation for the Hamming distance of the final round $10 \rightarrow 11$ is $\rho = 0.00532594$. We performed a CPA key recovery using 16 M traces and successfully recovered 12 of the 16 bytes of the full key. The remaining four bytes of the key can then be found in negligible time through exhaustive search with 2^{32} AES invocations. Incidentally, we note that the key recovery specifically fails for key bytes 0, 4, 8, and

12, *i.e.*, the first byte of each 4-byte word. This implies that these bytes might exhibit a different leakage behavior than the other (successfully recovered) bytes. Hence, with an appropriate leakage model, it might be possible to also directly recover those four bytes without exhaustive search. We leave this aspect for future work.

5.2.3. Kernel Module

Likewise, to our attack on the SGX enclave (Section 5.2.2), we evaluate the CPA attack on a Linux kernel module, processing an AES-NI key.

Setup. We implemented a kernel module encrypting data using AES-NI accelerated encryption. Therefore, we made use of the Intel AES-NI Sample Library [22] that claims to be some of the most efficient AES assembler code implementations [38]. The kernel module provides an `ioctl` interface to user space where data to be encrypted can be passed to.

Profiling. For the attack on the kernel module, we recorded 4 M traces ($\rho_{noise} = 0.0002$) in 50 h on the Xeon E3-1240 v5 (server) system. Each leakage sample corresponds to 16 384 encryptions of 16 kB in a loop inside the `ioctl` handler, similar to scenario 1 for SGX above. As for SGX, we observe statistically significant leakage for the AES rounds using both the Hamming weight and Hamming distance models. The profiling results are given in Table 9.8 in Section C.

Key Recovery. For the attack on the kernel module, we performed a CPA key recovery using the 4 M traces, again targeting the final round of AES. We successfully recovered 15 of the 16 bytes of the full key. Note that the correct candidate for the remaining byte was the second-best candidate.

5.2.4. Limitations

We showed that it is possible to recover secrets from AES-NI, both from implementations in the kernel and from an Intel IPP function using AES-NI in Intel SGX. These attacks are feasible, and the number of traces is also well in the threat model. For example, previous side-channel attacks

on AES-NI with physical access required recording a large number of traces for 17 days using an EM probe [70]—longer than the time required for our method. Furthermore, as input to the NISTIR 8214A draft [7], Rijmen and Svetla [69] recommend considering an adversary that can collect up to 100 M traces.

While we note that our attacks might succeed with fewer traces using algorithms designed to perform a CPA-guided exhaustive search [83], we did not evaluate that in our attacks. Still, whether our CPA attacks are practical depends on the target, as we require large amounts of data to be processed with a fixed or known plaintext. In the case of Intel SGX, as a privileged attacker, it might be possible to alleviate this issue using zero-stepping (see Section 4.2.2). Instead of repeating the whole algorithm, it is possible to repeat only the target instruction (which should also result in a better SNR). However, in our experiments so far, we could not successfully apply CPA in this case. This might be due to the noise introduced by the zero-stepping logic, combined with long measurement times, which prevent the acquisition of a sufficient number of traces. Finally, determining the appropriate leakage model depends on the specific implementation of the algorithm under attack and also the targeted CPU—e.g., we observed substantial differences for AES-NI between our i3-7100U and Xeon E3-1240 v5 systems, with the i3-7100U exposing less leakage (see Section C). We leave a more in-depth study of the behavior for future work.

5.3. Observing Intra-Cacheline Activity

A common assumption for side-channel-secure software was that an attacker can only observe victim operations at a cache line granularity [9]. For instance, to protect against cache attacks that observe access patterns at a cache line granularity, such as *Flush+Reload* and *Prime+Probe*, *scatter-gather* [24] is a constant-time programming technique for RSA. However, recent work [59, 88] showed that this assumption does not hold when an attacker shares a hyperthread with a victim. Consequently, the attacker can infer the cryptographic key used by an implementation that has sub-cache-line variations in the control flow or data accesses.

However, for our attack, we assume a scenario where the victim and attacker do not share a hyperthread. Consequently, previous attacks [59, 88] cannot obtain this information.

In our experiment, the victim performs a secret-dependent branch within a cache line, executing instructions with different power consumption. If the bit at a given offset of a secret byte is set, a `fscale` instruction is executed. Otherwise, `rdrand` is executed. We assume an unprivileged attacker that can trigger the code executed by the victim through an API passing the offset to it. We evaluated the experiment on our i7-8650U and i7-6600U (mobile) systems, both running on battery and connected to an AC power supply, both desktop machines (i7-6700K and i9-9000K) as well as on the 3 servers (E3-1240 v5, E3-1275 v5, Silver 4214). The attacker records the power consumption when triggering the victim. As illustrated in Figure 9.9, one can clearly distinguish jump-target locations within a cache line due to the difference in power consumption. Hence, constraining control-flow variations in cryptographic operations to a cache line cannot be considered secure anymore, even in scenarios where victim and attacker do not share a hyperthread. This allows breaking cryptographic implementations, which are currently considered secure in the scenario we investigate [31].

In addition, an extreme approach suggested to impede cache timing attacks is to disable caching for the `PRM` range in SGX [16]. In a second experiment, we mark pages of our victim as uncacheable. Thus, the code cannot leak through cache timings anymore. Still, with our power side-channel, we can observe the leakage.

5.4. Kernel Address Space Derandomization

In this section, we show that an unprivileged attacker can derandomize the kernel address space using RAPL. As there is no distinction between committed and non-committed instructions at the voltage regulator level, the power consumption also changes for transient instructions. Transient instructions are instructions that have been executed by an out-of-order processor but are never committed to the architectural state, e.g., instructions causing a fault [52] or instructions following a misspeculated branch [47]. The general concept of derandomizing the kernel address space is to distinguish between the transient access of mapped and unmapped kernel addresses via differences in power consumption. The current KASLR implementation randomly chooses one out of 512 2 MB-aligned virtual addresses as the base address for the entire kernel [71]. Hence, as the kernel binary itself does not support fine-grained randomization, knowing the base offset of the kernel allows to calculate the loca-

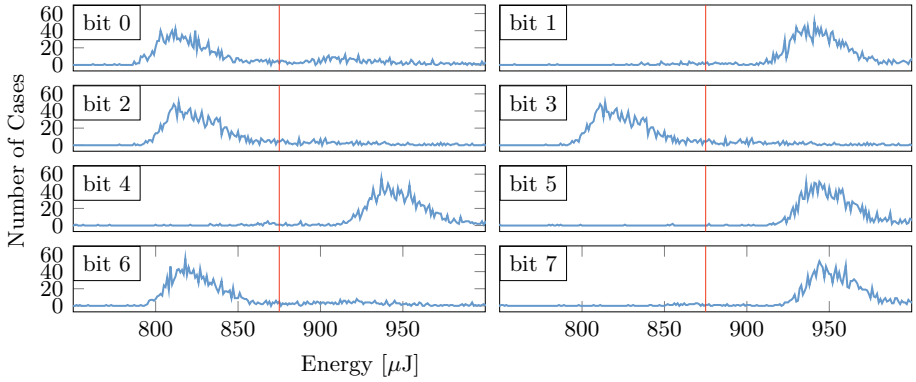


Figure 9.9.: Our attack clearly distinguishes different jumps within the same cache line. In this figure, leaking the byte `0x4d` (ASCII 'M') (`01001101` in binary) bit by bit by inspecting the power consumption. Values below that threshold are interpreted as '1's, values above as '0's.

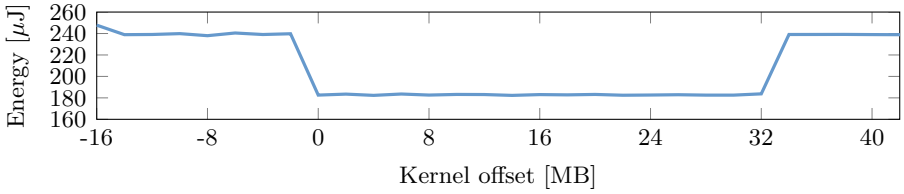


Figure 9.10.: Power consumption when transiently accessing kernel addresses. If a kernel page is not mapped, the access triggers an entire page-table walk, consuming more power.

tion of kernel code and data [11, 28, 36, 45, 71]. The same approach can also be applied to dynamically loadable kernel modules [71].

Transiently accessing mapped and unmapped kernel addresses show differences in timing [28, 36, 45] and store-forwarding behavior [11, 71]. Hence, the assumption is that there is also a measurable difference in power consumption.

Figure 9.10 shows the power consumption when transiently loading a kernel address while suppressing faults using Intel TSX. The power consumption differs for mapped and unmapped kernel pages. The differences in power consumption correlate with the differences in access times reported by Jang et al. [45]. As unmapped kernel pages cannot be cached

in the TLB, accessing these pages triggers a page-table walk, which consumes more power than accessing mapped kernel pages, which are cached in the TLB.

In our experiments, we used our i7-8650U (connected to a power supply and running on battery), i7-6600U, i9-9900K, and Xeon Silver 4214 systems with PTI (Page Table Isolation) disabled. Note that both, the i9-9900K and the Silver 4214, contain hardware mitigations against Meltdown; thus, PTI can be disabled. To evaluate the success rate, we execute the KASLR break 500 times for known KASLR offsets. On average, we successfully derandomize the KASLR offset in 100 % ($n = 500$, $\sigma_{\bar{x}} = 0.00$) of the runs. The average time to find the KASLR offset is 20 s. Hence, while not being the fastest KASLR break, it is still practical. Moreover, in contrast to previous microarchitectural KASLR breaks [11, 12, 28, 36, 45, 71], our KASLR break using power consumption is the first microarchitectural KASLR break, which does not require any timing primitive. Even with the microcode patch on the i9-9900K, there is no significant change in the success rate of the KASLR break. This is in line with Intel's statement that attacks on KASLR are not mitigated by this update [40].

In addition, we evaluated the influence of system activity using `stress-ng` on the success rate of the KASLR break on the i9-9900K running Ubuntu 18.04. These tests are designed to stress the CPU and do not represent a realistic workload, e.g., compilation task, rendering process, or office workload. However, the tool allows us to vary the load on each core. By default, it will cycle through all stress tests unless a specific one is specified. With a load below 10 % on the entire system, there is no change in the success rate. With a moderately high load of 50 %, it decreases to 22 % ($n = 100$, $\sigma_{\bar{x}} = 4.34$). However, as system noise is statistically independent from the measured signal, increasing the number of measurements (and thus the runtime) increases the success rate. Especially as system activity only *increases* the power consumption, and mapped pages have a *lower* power consumption than unmapped pages, noise does not lead to false positives, but only to not being able to detect the kernel (false negative). A simply increase of the measurements by a factor of 10 already results in a success rate of 46 % ($n = 100$, $\sigma_{\bar{x}} = 4.75$).

5.5. Timing-Independent Covert Channel

In this section, we describe how unprivileged access to power consumption can be utilized to establish a timing-independent covert channel. The

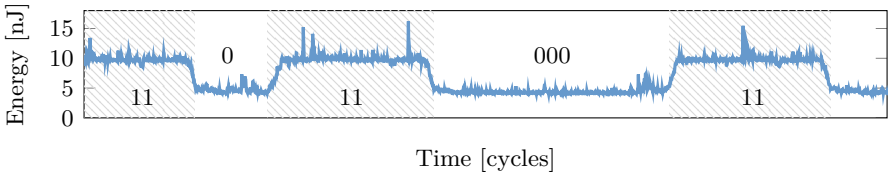


Figure 9.11.: Transmission of bits `1101100011` using the time-less covert channel.

basic idea of the covert channel is to encode the information by varying the power consumption of the device. To send a 1-bit, the sender increases the power consumption by executing more energy-consuming instructions. To transmit a 0-bit, the sender idles. The receiver monitors the power consumption of the device through the RAPL interface and decodes the transmitted information by observing the changes in power consumption.

Figure 9.11 illustrates the transmission of the bits `1101100011` over the power-based covert channel. We transmitted 1 kB of random data between two unprivileged processes running on different cores of the i7-8650U, either battery-powered or connected to a power supply. We achieved a transmission rate of 18.7 bit/s with a bit error rate of 0.89 %.

While the transmission rate of our covert channel is significantly lower in contrast to other state-of-the-art covert channels [29, 53, 57], our covert channel has the benefit that it does not rely on high-resolution timers. Furthermore, our proof-of-concept covert channel is not optimized and strictly working only with binary decisions. However, we can transmit not just one bit per symbol but rather several bits by using modulation techniques, such as amplitude modulation, phase-shift keying, or frequency modulation. While Maurice et al. [57] found that these methods are infeasible for cache covert channels due to the unreliable clock, they are applicable to a power-based covert channel. Thus, we believe that the performance of our covert channel could be drastically improved using these techniques.

6. Countermeasures

In this section, we discuss different countermeasures and mitigation strategies for the presented attacks.

Restricting Access. To obtain the Intel RAPL counters, kernel privileges are required to read the corresponding MSRs. However, the power capping framework `powercap` on Linux provides unprivileged access to these MSRs through the `sysfs` interface. While the purpose of the driver is to expose RAPL for user-space consumption [65], unprivileged access could be directly prevented by respecting the access level similar to `kernel.perf_event_paranoia` for the `perf` interface. While these interfaces may be required for existing functionality, limiting user-space access is necessary to mitigate at least unprivileged attacks. However, as a privileged attacker has direct access to these MSRs, attacks on Intel SGX are not prevented. Thus, access to these MSRs needs to be blocked via a microcode update. Furthermore, trusted computing base recovery is required to allow remote verifiers to re-establish the trust that these MSRs have been deactivated.

Limiting Resolution. The RAPL interface has a µJ resolution. While reducing the counter's granularity does not completely mitigate our attacks, the number of traces for some scenarios might become impractical. However, even without the RAPL interface, it may still be possible to use other limited-resolution sources of energy data, e.g., battery monitoring, to conduct a software-based power side-channel attack, e.g., identifying running applications [87].

Limiting Precise Execution Control. Restricting the user-space access to the RAPL counters only impedes unprivileged attackers, as a privileged attacker has direct access to these MSRs. In addition, the attacker can make use of precise execution control (cf. Section 4.2.2) to zero step an enclave. This primitive gives an attacker the possibility to execute a single instruction within an SGX enclave arbitrarily often, enabling sampling of the instruction's energy consumption (cf. Section 5.1). Introducing a counter inside SGX that increments every time an enclave is executed from the same instruction pointer could limit the number of zero steps.

Application Hardening. Software computing on particularly sensitive values, e.g., cryptographic algorithms, could deploy state-of-the-art countermeasures against power analysis, e.g., masking, to make these attacks more difficult. However, using zero stepping (Section 4.2.2) and the possibility to observe the Hamming weight of bytes (Section 3.5), masking is

insufficient against our attacks on SGX enclaves.

Intel's Mitigation. To address the presented issues, Intel released microcode updates that help ensure that the reported energy consumption by the RAPL interface hinders the ability to distinguish same instructions with different data or operands if SGX is enabled [40]. In addition, an update to the Linux `powercap` driver restricts the unprivileged access to the RAPL MSRs.

7. Related Work & Discussion

In this section, we present related and future work and discuss other microarchitectures.

7.1. Related Work

Hardware-based Power Analysis. Eisenbarth et al. [18] reconstructed control-flow and program code from power consumption on a small microcontroller. Strobel et al. [76] distinguish instructions on a microcontroller using an oscilloscope sampling at 2.5 GHz. Park et al. [66] use an oscilloscope with 2.5 GHz combined with machine learning to extract the instruction stream (opcodes and operands) from a microcontroller. Msgna et al. [61] measured differences in power consumption during the execution of single instructions on a microcontroller using an oscilloscope with a sampling rate of 5 GHz. Saab et al. [70] extracted an AES-NI key from an Intel i7 after collecting traces for 17 days with an EM probe.

Guri et al. [32], as well as Islam and Ren [44] demonstrated that current and voltage, respectively, can be monitored and influenced to build covert channels, e.g., in cloud environments. However, both works assume an attacker with hardware equipment connected to the device.

Undersampling. Molka et al. [60] used a physically-connected power meter to record a victim system's power consumption at a rate of 10 Hz, distinguishing loops of `nop`s and other instructions. Attacks with similar sampling rates to ours were shown by Genkin et al. [21], who recovered 4096-bit GnuPG RSA keys and program code via acoustic cryptanalysis, and Lifshits et al. [51], who inferred sensitive data, including keystrokes,

via a malicious battery storing power traces. These works sampled at ≈24 kHz (mobile phone attack) and 1 kHz, respectively.

Our work shows that this can similarly be done from software at even higher sampling rates, and our attacks demonstrate the security ramifications of this. While prior attacks require either physical proximity or physical access to the device, they support this work's finding that a low sampling rate can still achieve fine-grained information leakage.

Software-based Power Analysis. Fusi [19] used RAPL to attack RSA-16384 but concluded that the sampling rate of RAPL is too low to mount an attack, showing that it is only observable whether branches are taken, and accessed data is cached. Mantel et al. [55] distinguish RSA keys with different Hamming weights using RAPL but do not try to extract keys or perform other concrete attacks. Gao et al. [20] use RAPL in containers to infer information about the host environment, e.g., co-location of multiple containers.

Power Analysis on Mobile Devices. Yan et al. [87] monitor system power information on mobile devices to acquire voltage and current, observing a correlation with keystrokes, enabling them to infer password lengths and also distinguish different applications. Qin et al. [68] use the same interfaces to fingerprint websites on mobile devices. We instead use RAPL on regular laptops, desktops, and servers that have more subtle variation in power consumption and voltage.

On-die Power Analysis. O'Flynn [63] recorded power measurements using an on-board ADC from the non-secure world to recover secrets processed in the secure world on TrustZone-M. Zhao and Suh [89] use an FPGA to observe a CPU's power consumption on the same SoC to break RSA.

7.2. Other Microarchitectures

While we focus on Intel's RAPL implementation throughout this work, other microarchitectures offer different interfaces to obtain the energy consumption of the core.

For instance, since the Zen microarchitecture, AMD CPUs also provide a RAPL interface [3]. In contrast to Intel, their counters even allow to measure the energy consumption even per individual core. However, as the `powercap` driver does not support AMD's implementation, an attacker requires kernel privileges to read the corresponding MSRs. In Section A, we show that AMD's RAPL interface allows to distinguish different instructions executed on an AMD Ryzen CPU. This could allow similar attacks on AMD CPUs, e.g., against AMD's SEV-SNP, where a privileged kernel-space attacker is conceivable.

Other CPU manufacturers, e.g., ARM, NVIDIA, IBM POWER, Ampere, Hygon, or Marvell, provide different power interfaces as well. We briefly discuss them in Section A and leave the investigation of them to future work.

7.3. Enclave Inspection

While Intel SGX provides integrity and confidentiality of data and integrity of code at runtime, it does not provide confidentiality of code in the binary file stored offline. However, with the Intel Software Guard Extensions Protected Code Loader (Intel SGX PCL) [43], the enclave shared object is encrypted at build time and decrypted during the load phase. This enables intellectual property within SGX enclave code to be protected from inspection by untrusted parties, as reverse-engineering of the encrypted enclave is not possible [5]. Furthermore, encrypting the memory used by the enclave [16] prevents runtime inspection, provided the enclave is built in release mode [42].

Using zero-stepping, we can now measure the energy consumption of every single instruction executed within an SGX enclave. This allows to classify different instructions by evaluating their power consumption, as shown in Section 3. Further, differences depending on the values of their operands and loaded data from the cache can be observed. This enables us to not only recover the control flow of the executed program but also to directly disclose sensitive information, as we demonstrate in Section 5.1.

For enclave inspection, the idea is to retrofit the power-side-channel-based disassembler by Eisenbarth et al. [18] with PLATYPUS to infer the control flow of the enclave. While our results are promising for a certain set of instructions (see Table 9.3), the general case is very complex due to the complex instruction-set architecture. In total, there are more

than 3684 x86-64 instruction variants (combining mnemonics and operand types) [34] that need to be profiled on the microarchitecture under attack first. Thus, the set of instructions with similar power consumption, especially with the influence of different operand values, is currently too large. We leave further exploration to future work.

8. Conclusion

In this work, we show that software-based power side-channel attacks are particularly powerful against Intel SGX due to the zero-stepping capabilities of a privileged attacker. We showed how instructions and operand-level differences can be observed, enabling recovery of an RSA key from mbed TLS inside an SGX enclave. We demonstrated that with sufficient statistical evaluation, even user space attackers can exploit unprivileged access to the Intel RAPL interface to extract AES-NI keys from SGX enclaves or kernel space. Moreover, we demonstrated that this side channel enables an attacker to break KASLR, observe sub-cache-line-granularity activity, and establish timing-independent covert channels.

While unprivileged attacks can be impeded by restricting access to the `sysfs` interface, mitigating privileged attacks in order to protect Intel SGX enclaves is not trivial. We, therefore, propose limiting precise execution control and, while it, unfortunately, breaks backward compatibility and support for software-based thermal management, removing access to these interfaces in general.

Acknowledgments

We want to thank Peter Pessl (Infineon Technologies), Martin Haubenwallner, Martin Schwarzl (Graz University of Technology) and Stefan Mangard (Graz University of Technology).

The research presented in this paper was supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria, and Carinthia. It was also supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). It has also been supported by the Austrian Research

Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. It is partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/R012598/1, EP/S030867/1 and by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM).

Additional funding was provided by generous gifts from Intel, ARM, Amazon, and Red Hat. Further, we would like to thank Equinix Metal for providing us access to bare metal instances to run our experiments.

Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] National Security Agency. *TEMPEST: A Signal Problem*. 1972.

[2] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. "Amplifying Side Channels Through Performance Degradation". In: *ACSAC*. 2016.

[3] *AMD uProf User Guide*. 3.2. Advanced Micro Devices Inc. 2019.

[4] ARM. *mbed TLS*. 2020. URL: `https:///tls.mbed.org`.

[5] Jean-Philippe Aumasson and Luis Merino. "SGX Secure Enclaves in Practice: Security and Crypto Review". In: *Black Hat Briefings*. 2016.

[6] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. "Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures". In: *HPCA*. 2013.

[7] Lús TAN Brandão, Michael Davidson, and Apostol Vassilev. *Towards NIST Standards for Threshold Schemes for Cryptographic Primitives: A Preliminary Roadmap*. Tech. rep. National Institute of Standards and Technology, 2019.

[8] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *WOOT*. 2017.

[9] Ernie Brickell. *Technologies to Improve Platform Security*. CHES. 2011.

[10] Eric Brier, Christophe Clavier, and Francis Olivier. “Correlation Power Analysis with a Leakage Model”. In: *CHES*. 2004.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. “Fallout: Leaking Data on Meltdown-resistant CPUs”. In: *CCS*. 2019.

[12] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. “KASLR: Break It, Fix It, Repeat”. In: *AsiaCCS*. 2020.

[13] A. P. Chandrakasan and R. W. Brodersen. “Minimizing power consumption in digital CMOS circuits”. In: *Proceedings of the IEEE* (1995).

[14] Ampere Computing. *Ampere AltraTM Linux Kernel Porting Guide*. 2020. URL: `https://github.com/AmpereComputing/ampere-centos-kernel/wiki/Ampere-AltraTM-Linux-Kernel-Porting-Guide`.

[15] Intel Corporation. *What exactly is a P-state? (Pt. 1)*. 2015.

[16] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *Cryptology ePrint Archive, Report 2016/086* (2016).

[17] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX”. In: *USENIX Security Symposium*. 2017.

[18] Thomas Eisenbarth, Christof Paar, and Björn Weghenkel. “Building a Side Channel Based Disassembler”. In: *Transactions on computational science X*. Springer, 2010.

[19] Matteo Fusi. *Information-Leakage Analysis Based on Hardware Performance Counters*. 2017.

[20] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. “ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds”. In: *DSN*. 2017.

[21] Daniel Genkin, Adi Shamir, and Eran Tromer. “Acoustic Cryptanalysis”. In: *Journal of Cryptology* (2017).

[22] Gladman, Brian. *Intel AESNI Sample Library*. 2013. URL: `https://software.intel.com/content/www/us/en/develop/articles/download-intel-aesni-sample-library.html`.

[23] Jovan D Golić and Christophe Tymen. “Multiplicative Masking and Power Analysis of AES”. In: *CHES*. 2002.

[24] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. “Fast and Constant-Time Implemen-

tation of Modular Exponentiation". In: *Embedded Systems and Communications Security* (2009).

[25] Louis Goubin and Jacques Patarin. "DES and Differential Power Analysis: The "Duplication" Method". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. 1999.

[26] Corey Gough, Ian Steiner, and Winston Saunders. *Energy Efficient Servers*. Apress, 2015.

[27] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory". In: *USENIX Security Symposium*. 2017.

[28] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". In: *CCS*. 2016.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *DIMVA*. 2016.

[30] Amina Guermouche and Anne-Cécile Orgerie. *Experimental analysis of vectorized instructions impact on energy and power consumption under thermal design power constraints*. 2019.

[31] Shay Gueron. *Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01*. 2012.

[32] Mordechai Guri, Boris Zadov, Dima Bykhovsky, and Yuval Elovici. "PowerHammer: Exfiltrating Data from Air-Gapped Computers Through Power Lines". In: *arXiv:1804.04014* (2018).

[33] Andreas Herrmann. *Kernel driver fam15h_power: The Linux Kernel documentation*. 2019. URL: `https://www.kernel.org/doc/html/v5.4-preprc-cpu/hwmon/fam15h%5C_power.html`.

[34] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stratified Synthesis: Automatically Learning the x86-64 Instruction Set". In: *PLDI*. ACM, 2016.

[35] Mikael Hirki, Zhonghong Ou, Kashif Nizam Khan, Jukka K Nurminen, and Tapio Niemi. "Empirical study of the power consumption of the x86-64 instruction decoder". In: *USENIX CoolDC*. 2016.

[36] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR". In: *S&P*. 2013.

[37] IBM. *POWER9 Processor User's Manual*. 2.0. 2018.

[38] Intel. *Advanced Encryption Standard (AES) Crypto Performance Analysis Project*. 2013.

[39] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. 2019.

[40] Intel. *Intel-SA-00389*. 2020. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html.

[41] Intel. *Intel® Integrated Performance Primitives*. 2020. URL: https://software.intel.com/content/www/us/en/develop/tools/integrated-performance-primitives.html.

[42] Intel Corporation. *Intel SGX: Debug, Production, Pre-release – What's the Difference?* Jan. 2016.

[43] Intel Corporation. *Intel Software Guard Extensions (Intel SGX) Protected Code Loader (PCL) for Linux*. May 2018.

[44] Mohammad A Islam and Shaolei Ren. "Ohm's Law in Data Centers: A Voltage Side Channel for Timing Power Attacks". In: *CCS*. 2018.

[45] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX". In: *CCS*. 2016.

[46] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ToMPECS* (2018).

[47] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P*. 2019.

[48] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. "Introduction to Differential Power Analysis". In: *Journal of Cryptographic Engineering* (2011).

[49] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *CRYPTO'99*. 1999.

[50] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *USENIX Security Symposium*. 2017.

[51] Pavel Lifshits, Roni Forte, Yedid Hoshen, Matt Halpern, Manuel Philipose, Mohit Tiwari, and Mark Silberstein. "Power to peep-all: Inference Attacks by Malicious Batteries on Mobile Devices". In: *Proceedings on Privacy Enhancing Technologies* 2018.4 (2018).

[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security Symposium*. 2018.

[53] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical". In: *S&P*. 2015.

[54] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Science & Business Media, 2008.

[55] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. "How Secure is Green IT? The Case of Software-Based Energy Side Channels". In: *European Symposium on Research in Computer Security*. 2018.

[56] Marvell. *tx2mon*. 2020. URL: `https://github.com/Marvell-SPBU/tx2mon`.

[57] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: *NDSS*. 2017.

[58] Abdelhafid Mazouz, David C Wong, David Kuck, and William Jalby. "An Incremental Methodology for Energy measurement and Modeling". In: *ACM ICPE*. 2017.

[59] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "MemJam: A False Dependency Attack against Constant-Time Crypto Implementations in SGX". In: *CT-RSA*. 2018.

[60] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S Müller. "Characterizing the Energy Consumption of Data Transfers and Arithmetic Operations on x86-64 Processors". In: *International Conference on Green Computing*. IEEE. 2010.

[61] Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes. "Precise Instruction-Level Side Channel Profiling of Embedded Processors". In: *International Conference on Information Security Practice and Experience*. 2014.

[62] NVIDIA. *Jetson TX2: Thermal Design Guide*. 2017.

[63] Colin O'Flynn and Alex Dewar. "On-Device Power Analysis Across Hardware Security Domains". In: *CHES* (2019).

[64] *Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh*. 3.03. Advanced Micro Devices Inc. July 2018.

[65] Jacob Pan. *RAPL (Running Average Power Limit) driver*. 2013. URL: https://lwn.net/Articles/545745/.

[66] Jungmin Park, Xiaolin Xu, Yier Jin, Domenic Forte, and Mark Tehranipoor. "Power-based Side-Channel Instruction-level Disassembler". In: *DAC*. 2018.

[67] James Phung, Young Choon Lee, and Albert Y Zomaya. "Modeling System-Level Power Consumption Profiles Using RAPL". In: *NCA*. IEEE. 2018.

[68] Yi Qin and Chuan Yue. "Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7". In: *TrustCom/BigDataSE*. 2018.

[69] Vincent Rijmen and Svetla Nikova. *Threshold Cryptography Against Physical Attacks*. 2020. URL: https://www.esat.kuleuven.be/cosic/events/tis-online-workshop/wp-content/uploads/sites/6/2020/07/Vincent_Rijmen.pdf.

[70] Sami Saab, Pankaj Rohatgi, and Craig Hampel. "Side-Channel Protections for Cryptographic Instruction Set Extensions". In: *IACR Cryptology ePrint Archive* (2016).

[71] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs". In: *arXiv:1905.05725* (2019).

[72] Michael Schwarz, Daniel Gruss, Moritz Lipp, Maurice Clémentine, Thomas Schuster, Anders Fogh, and Stefan Mangard. "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features". In: *AsiaCCS* (2018).

[73] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks ". In: *DIMVA*. 2017.

[74] Yakun Sophia Shao and David Brooks. "Energy Characterization and Instruction-Level Energy Model of Intel's Xeon Phi Processor". In: *ISLPED*. 2013.

[75] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. "MicroScope: Enabling Microarchitectural Replay Attacks". In: *ISCA*. 2019.

[76] Daehyun Strobel, Florian Bache, David Oswald, Falk Schellenberg, and Christof Paar. "SCANDALee: A Side-ChANnel-based DisAssembLer using Local Electromagnetic Emanations". In: *DATE*. 2015.

[77] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. "Power Analysis of Embedded Software: A First Step towards Software Power Minimization". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (1994).

[78] Unified Extensible Firmware Interface (UEFI) Forum. *Advanced Configuration and Power Interface (ACPI) Specification, Version 6.3*. 2019.

[79] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security Symposium*. 2018.

[80] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *S&P*. 2020.

[81] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Workshop on System Software for Trusted Execution*. 2017.

[82] Evangelos Vasilakis. "An Instruction Level Energy Characterization of ARM Processors". In: *FORTH-ICS/TR-450* (2015).

[83] Nicolas Veyrat-Charvillon, Benôt Gérard, Mathieu Renauld, and François-Xavier Standaert. "An Optimal Key Enumeration Algorithm and its Application to Side-Channel Attacks". In: *SAC*. 2012.

[84] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. "Differentiating Code from Data in x86 Binaries". In: *ECML PKDD*. 2011.

[85] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves". In: *ESORICS*. 2016.

[86] Pu Wen. *Add support for Hygon Fam 18h (Dhyana) RAPL*. 2019. URL: `https://patchwork.kernel.org/patch/11123607/`.

[87] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. "A Study on Power Side Channels on Mobile Devices". In: *Symposium on Internetware*. 2015.

[88] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *JCEN* (2017).

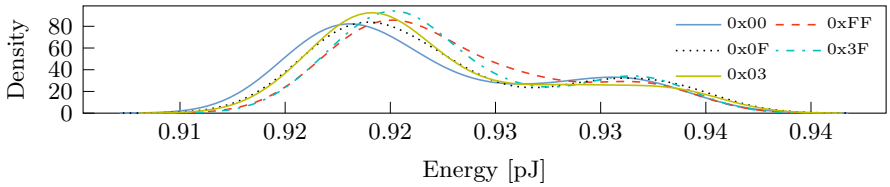


Figure 9.12.: Measured energy consumption of the `shr` instruction with different Hamming weights (AMD Ryzen 5 3600).

[89] Mark Zhao and G Edward Suh. "FPGA-based Remote Power Side-Channel Attacks". In: *S&P*. 2018.

Appendix

A. Other Microarchitectures

While we focus on Intel's RAPL implementation throughout this work, other microarchitectures offer different interfaces to obtain the energy consumption of the core.

AMD. Since the Zen (family 17H) microarchitecture, AMD CPUs have provided a RAPL interface [3]. However, little documentation is available regarding its implementation. Power consumption values for the core and package domains are provided respectively in the `CORE_ENERGY_STAT` and `PKG_ENERGY_STAT` MSRs [64]. A notable difference from Intel RAPL is that the core domain is accessible per-core rather than across all cores of the CPU, which substantially reduces measurement noise.

An attacker targeting AMD currently requires root privileges to read the MSRs, as the Linux `powercap` driver only establishes a user-accessible `sysfs` RAPL interface on Intel CPUs. While they lack the RAPL interface, earlier AMD family 15h and 16h CPUs also have power MSRs which provide consumption estimates based on platform activity levels [3]. Unfortunately, none were available to us for evaluation purposes. However, we believe systems with these CPUs may be vulnerable to user-space attacks because if the `fam15h_power` driver is loaded, power values can be read from user space via `sysfs` [33].

Table 9.5 lists the measured energy consumption of different instructions on AMD Ryzen 7 Pro 3700U, AMD Ryzen 7 3700X, and AMD EPYC

Instruction	Ryzen 7 Pro 3700U	Ryzen 7 3700X	EPYC 7401P
`nop`	0.0886 nJ	0.1052 nJ	0.1571 nJ
`inc r64`	0.1241 nJ	0.1144 nJ	0.1800 nJ
`xor r64, r64`	0.1246 nJ	0.1144 nJ	0.1785 nJ
`mov r64, mem`	0.0978 nJ	0.1266 nJ	0.1571 nJ
`imul r64, r64`	0.0930 nJ	0.0930 nJ	0.1586 nJ
`fscale`	0.0892 nJ	0.0991 nJ	0.1571 nJ
`rdrand r64`	0.0669 nJ	0.0564 nJ	0.0991 nJ
`rdtsc`	0.0885 nJ	0.0896 nJ	0.1296 nJ
`clflush mem`	0.0671 nJ	0.0503 nJ	0.0991 nJ
`aesenc xmm, xmm`	0.0890 nJ	0.0854 nJ	0.1571 nJ

Table 9.5.: Average observed energy consumption (package domain) of different instructions on an AMD Ryzen 7 Pro 3700U mobile CPU, an AMD Ryzen 7 3700X desktop CPU, and an AMD EPYC 7401P server CPU.

7401P. On the mobile and desktop CPU, we disabled processor boost and set the cores to a fixed frequency. To measure the energy consumption of an instruction, we record its energy consumption over 10 000 consecutive executions and take the median value to eliminate system-level noise (e.g., erroneous high values caused by interrupt handling or the process being descheduled). On both AMD and Intel (see Section 3), we observe the energy consumption across the entire CPU package to ensure that non-core activity (for example, interactions with DRAM) is included. We can observe inter-instruction differences in energy consumption. This enables identification of executed instructions, provided the attacker can profile the energy consumption of the victim microarchitecture. Furthermore, as shown in Figure 9.12, the Hamming weight of the register influences the energy consumption of the `shr` instruction.

ARM. The ARM Energy Probe, a 3-channel USB voltmeter which can be attached to a targeted platform, requires physical access to the device. However, different development boards using ARM CPUs contain on-board energy meters like the ARM CoreTile Express A15x2. The odroid XU+E used by Vasilakis [82] to characterize the energy consumption of instructions on ARM contains 4 `ina23` power sensors. The SAML11 running a Cortex-M23 processor used by O'Flynn and Dewar [63], grants access to an onboard ADC.

NVIDIA. NVIDIA's JetsonTX2 module has 3-channel INA3221 monitors [62] exposing current (mA), voltage (mV), and power (mW) used of different power rails. These include the CPU and GPU and are exposed to unprivileged access in the `sysfs`.

IBM POWER. The POWER9 processor contains a dedicated on-chip microcontroller that allows to analog sample various voltage rails. Note, however, that the POWER9 does not include per-core power estimation circuitry [37].

Marvell. For the ThunderX2, Marvell provides a kernel driver [56] exposing readings from hardware sensors, among other things, voltages and power measurements. Similar to Intel RAPL, measurements can be observed for all cores on the System on Chip, the SRAM, memory, and miscellaneous peripherals.

Ampere. For the Ampere Altra SoC, the APM X-Gene SoC hardware monitoring driver gives unprivileged access to the temperature and power sensors and, thus, allows to read the current power consumption of the CPU or the IO [14].

Hygon. Recently, RAPL support for the Hygon Dhyana CPU family has been added to the Linux `perf` interface and, likewise to AMD, allows to read the per-core energy consumption [86].

B. mbed TLS Attack

Figure 9.13 illustrates the minimum core voltage measurements for each key bit instruction of the mbed TLS attack described in Section 5.1.

C. Additional Profiling Results for AES-NI

Table 9.6, Table 9.7 and Table 9.8 present additional profiling correlations for AES-NI of the attacks described in Section 5.2.

Table 9.6.: Profiling correlations (for Xeon E3-1240 v5) after 16 M traces for AES-NI in scenario 2 for the Hamming weight (HW) for each round and Hamming distance (HD) between rounds. Bold entries and a $|\mathrm{SF}| \geq 1$ highlight significant statistical dependencies.

HD	ρ	**SF**	**HW**	ρ	**SF**
00 → 01	0.01412518	**14**	00	0.06653038	**66**
01 → 02	0.00674140	**6.7**	01	0.01389394	**14**
02 → 03	0.01182713	**12**	02	0.00045177	0.45
03 → 04	0.01159959	**12**	03	0.00106697	**1.1**
04 → 05	0.01144089	**11**	04	0.00073025	0.73
05 → 06	0.01069259	**11**	05	0.00058525	0.58
06 → 07	0.01142695	**11**	06	0.00114676	**1.1**
07 → 08	0.01158716	**12**	07	0.00068475	0.68
08 → 09	0.01102899	**11**	08	0.00077455	0.77
09 → 10	0.01114280	**11**	09	0.00094852	0.95
10 → 11	0.00532594	**5.3**	10	-0.00041563	-0.41
			11	0.05861710	**58**

Table 9.7.: Profiling correlations (for i3-7100U) after 4 M traces for AES-NI in scenario 1 for the Hamming weight (HW) for each round and Hamming distance (HD) between rounds. Bold entries and a $|\mathrm{SF}| \geq 1$ highlight significant statistical dependencies.

HD	ρ	**SF**	**HW**	ρ	**SF**
00 → 01	0.00429156	**2.1**	00	0.00957385	**4.8**
01 → 02	0.00256447	**1.3**	01	0.00550198	**2.7**
02 → 03	0.00441708	**2.2**	02	0.00056316	0.28
03 → 04	0.00404454	**2**	03	0.00003843	0.01
04 → 05	0.00388573	**1.9**	04	0.00048580	0.24
05 → 06	0.00512078	**2.6**	05	0.00081453	0.41
06 → 07	0.00418470	**2.1**	06	-0.00057528	-0.29
07 → 08	0.00454403	**2.3**	07	-0.00040692	-0.2
08 → 09	0.00477473	**2.4**	08	-0.00005976	-0.03
09 → 10	0.00488921	**2.4**	09	0.00085888	0.43
10 → 11	0.00269663	**1.3**	10	0.00021935	0.11
			11	0.01133641	**5.7**

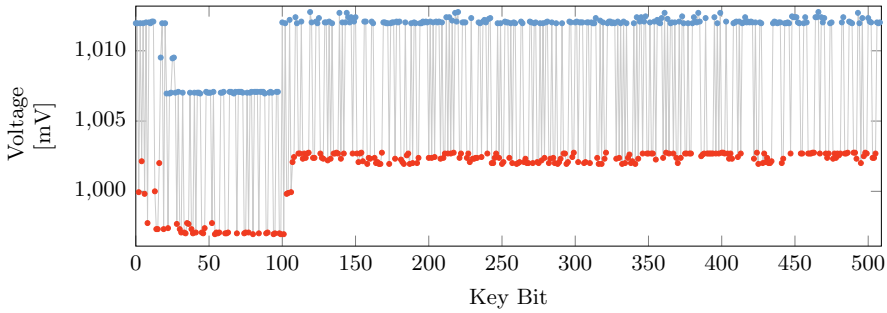


Figure 9.13.: Core voltage per measured instruction for each key bit offset in the fixed window length implementation of mbed TLS inside an SGX enclave on the Xeon E3-1275 v5. The blue marks represent 1 bits, while the red marks represent 0 bits. Using a threshold (dashed line), they can easily be distinguished.

Table 9.8.: Profiling correlations (for Xeon E3-1240 v5) after 4 M traces for AES-NI in the Linux kernel for the Hamming weight (HW) for each round and Hamming distance (HD) between rounds. Bold entries and a $|\mathrm{SF}| \geq 1$ highlight significant statistical dependencies.

HD	ρ	**SF**	**HW**	ρ	**SF**
00 → 01	0.063436878	**32**	00	0.092565061	**46**
01 → 02	0.029847718	**15**	01	0.075098846	**38**
02 → 03	0.056173544	**28**	02	0.0022803663	**1.1**
03 → 04	0.057817586	**29**	03	0.0033372879	**1.7**
04 → 05	0.057572691	**29**	04	0.0030430309	**1.5**
05 → 06	0.057020521	**28**	05	0.0034340331	**1.7**
06 → 07	0.058405015	**29**	06	0.0038034749	**1.9**
07 → 08	0.05697378	**28**	07	0.0022000058	**1.1**
08 → 09	0.057203062	**29**	08	0.0033568495	**1.7**
09 → 10	0.05837099	**29**	09	0.0031144225	**1.6**
10 → 11	0.027001464	**13**	10	-0.0008108201	-0.16
			11	0.12527739	**63**

10

KASLR is Dead: Long Live KASLR

Publication Data

Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is Dead: Long Live KASLR". in: *ESSoS*. 2017

Contributions

Large parts of the text and experiments.

KASLR is Dead: Long Live KASLR

Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner,
Clémentine Maurice Stefan Mangard

Graz University of Technology

Abstract

Modern operating system kernels employ address space layout randomization (ASLR) to prevent control-flow hijacking attacks and code-injection attacks. While kernel security relies fundamentally on preventing access to address information, recent attacks have shown that the hardware directly leaks this information. Strictly splitting kernel space and user space has recently been proposed as a theoretical concept to close these side channels. However, this is not trivially possible due to architectural restrictions of the x86 platform.

In this paper we present *KAISER*, a system that overcomes limitations of x86 and provides practical kernel address isolation. We implemented our proof-of-concept on top of the Linux kernel, closing all hardware side channels on kernel address information. *KAISER* enforces a strict kernel and user space isolation such that the hardware does not hold any information about kernel addresses while running in user mode. We show that *KAISER* protects against double page fault attacks, prefetch side-channel attacks, and TSX-based side-channel attacks. Finally, we demonstrate that *KAISER* has a runtime overhead of only 0.28%.

1. Introduction

Like user programs, kernel code contains software bugs which can be exploited to undermine the system security. Modern operating systems use hardware features to make the exploitation of kernel bugs more difficult. These protection mechanisms include making code non-writable and data non-executable. Moreover, accesses from kernel space to user space require additional indirection and cannot be performed through user space pointers directly anymore (SMAP/SMEP). However, kernel bugs can be exploited within the kernel boundaries. To make these attacks harder, address space layout randomization (ASLR) can be used to

make some kernel addresses or even all kernel addresses unpredictable for an attacker. Consequently, powerful attacks relying on the knowledge of virtual addresses, such as return-oriented-programming (ROP) attacks, become infeasible [14, 17, 19]. It is crucial for kernel ASLR to withhold any address information from user space programs. In order to eliminate address information leakage, the virtual-to-physical address information has been made unavailable to user programs [13].

Knowledge of virtual or physical address information can be exploited to bypass KASLR [7, 22], bypass SMEP and SMAP [11], perform side-channel attacks [5, 15, 18], Rowhammer attacks [6, 12, 20], and to attack system memory encryption [2]. To prevent attacks, system interfaces leaking the virtual-to-physical mapping have recently been fixed [13]. However, hardware side channels might not easily be fixed without changing the hardware. Specifically side-channel attacks targeting the page translation caches provide information about virtual and physical addresses to the user space. Hund et al. [7] described an attack exploiting double page faults, Gruss et al. [5] described an attack exploiting software prefetch instructions,[1] and Jang et al. [10] described an attack exploiting Intel TSX (hardware transactional memory). These attacks show that current KASLR implementations have fatal flaws, subsequently KASLR has been proclaimed dead by many researchers [3, 5, 10].

Gruss et al. [5] and Jang et al. [10] proposed to unmap the kernel address space in the user space and vice versa. However, this is non-trivial on modern x86 hardware. First, modifying page table structures on context switches is not possible due to the highly parallelized nature of today's multi-core systems, e.g., simply unmapping the kernel would inhibit parallel execution of multiple system calls. Second, x86 requires several locations to be valid for both user space and kernel space during context switches, which are hard to identify in large operating systems. Third, switching or modifying address spaces incurs Translation Lookaside Buffer (TLB) flushes [8]. Jang et al. [10] suspected that switching address spaces may have a severe performance impact, making it impractical.

In this paper, we present *KAISER*, a highly-efficient practical system for kernel address isolation, implemented on top of a regular Ubuntu Linux. *KAISER* uses a shadow address space paging structure to separate kernel

[1] The list of authors for "Prefetch Side-Channel Attacks" by Gruss et al. [5] and this paper overlaps.

space and user space. The lower half of the shadow address space is synchronized between both paging structures. Thus, multiple threads work in parallel on the two address spaces if they are in user space or kernel space respectively. *KAISER* eliminates the usage of global bits in order to avoid explicit TLB flushes upon context switches. Furthermore, it exploits optimizations in current hardware that allow switching address spaces without performing a full TLB flush. Hence, the performance impact of *KAISER* is only 0.28%.

KAISER reduces the number of overlapping pages between user and kernel address space to the absolute minimum required to run on modern x86 systems. We evaluate all microarchitectural side-channel attacks on kernel address information that are applicable to recent Intel architectures. We show that *KAISER* successfully eliminates the leakage in all cases.

Contributions. The contributions of this work are:

1. *KAISER* is the first practical system for kernel address isolation. It introduces shadow address spaces to utilize modern CPU features efficiently avoiding frequent TLB flushes. We show how all challenges to make kernel address isolation practical can be overcome.

2. Our open-source proof-of-concept implementation in the Linux kernel shows that *KAISER* can easily be deployed on commodity systems, *i.e.*, a full-fledged Ubuntu Linux system.[2]

3. After KASLR has already been considered dead by many researchers, *KAISER* fully restores the former efficacy of KASLR with a runtime overhead of only 0.28%.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background on kernel protection mechanisms and side-channel attacks. In Section 3, we describe the design and implementation of *KAISER*. In Section 4, we evaluate the efficacy of *KAISER* and its performance impact. In Section 5, we discuss future work. We conclude in Section 6.

[2]We are preparing a submission of our patches into the Linux kernel upstream. The source code and the Debian package compatible with Ubuntu 16.10 can be found at `https://github.com/IAIK/KAISER`.

2. Background

2.1. Virtual Address Space

Virtual addressing is the foundation of memory isolation between different processes as well as processes and the kernel. Virtual addresses are translated to physical addresses through a multi-level translation table stored in physical memory. A CPU register holds the physical address of the active top-level translation table. Upon a context switch, the register is updated to the physical address of the top-level translation table of the next process. Consequently, processes cannot access all physical memory but only the memory that is mapped to virtual addresses. Furthermore, the translation tables entries define properties of the corresponding virtual memory region, e.g., read-only, user-accessible, non-executable.

On modern Intel x86-64 processors, the top-level translation table is the page map level 4 (PML4). Its physical address is stored in the `CR3` register of the CPU. The PML4 divides the 48-bit virtual address space into 512 PML4 entries, each covering a memory region of 512 GB. Each subsequent level sub-divides one block of the upper layer into 512 smaller regions until 4 kB pages are mapped using page tables (PTs) on the last level. The CPU has multiple levels of caches for address translation table entries, the so-called TLBs. They speed up address translation and privilege checks. The kernel address space is typically a defined region in the virtual address space, e.g., the upper half of the address space.

Similar translation tables exist on modern ARM (Cortex-A) processors too, with small differences in size and property bits. One significant difference to x86-64 is that ARM CPUs have two registers to store physical addresses of translation tables (TTBR0 and TTBR1). Typically, one is used to map the user address space (lower half) whereas the other is used to map the kernel address space (upper half). Gruss et al. [5] speculated that this might be one of the reasons why the attack does not work on ARM processors. As x86-64 has only one translation-table register (`CR3`), it is used for both user and kernel address space. Consequently, to perform privilege checks upon a memory access, the actual page translation tables have to be checked.

Control-Flow Attacks. Modern Intel processors protect against code injection attacks through non-executable bits. Furthermore, code execution and data accesses on user space memory are prevented in kernel

mode by the CPU features supervisor-mode access prevention (SMAP) and supervisor-mode execution prevention (SMEP). However, it is still possible to exploit bugs by redirecting the code execution to existing code. Solar Designer [23] showed that a non-executable stack in user programs can be circumvented by jumping to existing functions within libc. Kemerlis et al. [11] presented the *ret2dir* attack which redirects a hijacked control flow in the kernel to arbitrary locations using the kernel physical direct mapping. Return-oriented programming (ROP) [21] is a generalization of such attacks. In ROP attacks, multiple code fragments—so-called gadgets—are chained together to build an exploit. Gadgets are not entire functions, but typically consist of one or more useful instructions followed by a return instruction.

To mitigate control-flow-hijacking attacks, modern operating systems randomize the virtual address space. Address space layout randomization (ASLR) ensures that every process has a new randomized virtual address space, preventing an attacker from knowing or guessing addresses. Similarly, the kernel has a randomized virtual address space every time it is booted. As Kernel ASLR makes addresses unpredictable, it protects against ROP attacks.

2.2. CPU Caches

Caches are small memory buffers inside the CPU, storing frequently used data. Modern Intel CPUs have multiple levels of set-associative caches. The last-level cache (LLC) is shared among all cores. Executing code or accessing data on one core has immediate consequences for all other cores.

Address translation tables are stored in physical memory. They are cached in regular data caches [8] but also in special caches such as the translation lookaside buffers. Figure 10.1 illustrates how the address translation caches are used for address resolution.

2.3. Microarchitectural Attacks on Kernel Address Information

Until recently, Linux provided information on virtual and physical addresses to any unprivileged user program through operating system interfaces. As this information facilitates mounting microarchitectural attacks, the interfaces are now restricted [13]. However, due to the way the

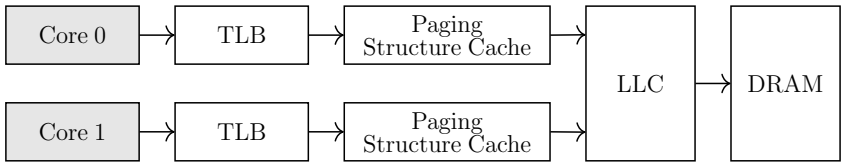


Figure 10.1.: Address translation caches are used to speed up address translation table lookups.

processor works, side channels through address translation caches [4, 5, 7, 10] and the branch-target buffer [3] leak parts of this information.

Address Translation Caches. Hund et al. [7] described a double page fault attack, where an unprivileged attacker tries to access an inaccessible kernel memory location, triggering a page fault. After the page fault interrupt is handled by the operating system, the control is handed back to an error handler in the user program. The attacker measures the execution time of the page fault interrupt. If the memory location is valid, regardless of whether it is accessible or not, address translation table entries are copied into the corresponding address translation caches. The attacker then tries to access the same inaccessible memory location again. If the memory location is valid, the address translation is already cached and the page fault interrupt will take less time. Thus, the attacker learns whether a memory location is valid or not, even if it is not accessible from the user space.

Jang et al. [10] exploited the same effect in combination with Intel TSX. Intel TSX is an extension to the x86 instruction set providing a hardware transactional memory implementation via so-called TSX transactions. If a page fault occurs within a TSX transaction, the transaction is aborted without any operating system interaction. Thus, the entire page fault handling of the operation system is skipped, and the timing differences are significantly less noisy. In this attack, the attacker again learns whether a memory location is valid, even if it is not accessible from the user space.

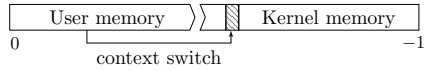
Gruss et al. [5] exploited software prefetch instructions to trigger address translation. The execution time of the prefetch instruction depends on which address translation caches hold the right translation entries. Thus, in addition to learning whether an inaccessible address is valid or not, an attacker learns its corresponding page size as well. Furthermore, software prefetches can succeed even on inaccessible memory. Linux has a kernel

physical direct map, providing direct access to all physical memory. If the attacker prefetches an inaccessible address in this kernel physical direct map corresponding to a user-accessible address, it will also be cached when accessed through the user address. Thus, the attacker can retrieve the exact physical address for any virtual address.

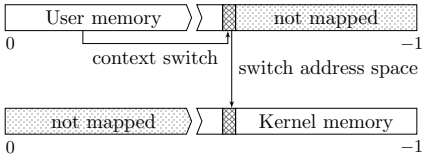
All three attacks have in common that they exploit that the kernel address space is mapped in user space as well, and that accesses are only prevented through the permission bits in the address translation tables. Thus, they use the same entries in the paging structure caches. On ARM architectures, the user and kernel addresses are already distinguished based on registers, and thus no cache access and no timing difference occurs. Gruss et al. [5] and Jang et al. [10] proposed to unmap the entire kernel space to emulate the same behavior as on the ARM architecture.

Branch-Target Buffer. Evtyushkin et al. [3] presented an attack on the branch-target buffer (BTB) to recover the lowest 30 bits of a randomized kernel address. The BTB is indexed based on the lowest 30 bits of the virtual address. Similar as in a regular cache attack, the adversary occupies parts of the BTB by executing a sequence of branch instructions. If the kernel uses virtual addresses with the same value for the lowest 30 bits as the attacker, the sequence of branch instructions requires more time. Through targeted execution of system calls, the adversary can obtain information about virtual addresses of code that is executed during a system call. Consequently, the BTB attack defeats KASLR.

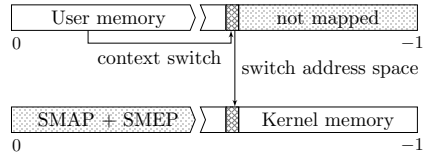
We consider the BTB attack out of scope for our countermeasure (*KAISER*), which we present in the next section, for two reasons. First, Evtyushkin et al. [3] proposed to use virtual address bits > 30 to randomize memory locations for KASLR as a zero-overhead countermeasure against their BTB attack. Indeed, an adaption of the corresponding range definitions in modern operating system kernels would effectively mitigate the attack. Second, the BTB attack relies on a profound knowledge of the behavior of the BTB. The BTB attack currently does not work on recent architectures like Intel Skylake, as the BTB has not been reverse-engineered yet. Consequently, we also were not able to reproduce the attack in our test environment (Intel Skylake i7-6700K).



(a) Regular OS



(b) Stronger kernel isolation



(c) KAISER

Figure 10.2.: (a) The kernel is mapped into the address space of every user process. (b) Theoretical concept of stronger kernel isolation. It splits the address spaces and only interrupt handling code is mapped in both address spaces. (c) For compatibility with x86 Linux, KAISER relies on SMAP to prevent invalid user memory references and SMEP to prevent execution of user code in kernel mode.

3. Design and Implementation of *KAISER*

In this section, we describe the design and implementation of *KAISER*[3]. We discuss the challenges of implementing kernel address isolation. We show how shadow address space paging structures can be used to separate kernel space and user space. We describe how modern CPU features and optimizations can be used to reduce the amount of regular TLB flushes to a minimum. Finally, to show the feasibility of the approach, we implemented *KAISER* on top of the latest Ubuntu Linux kernel.

3.1. Challenges of Kernel Address Isolation

As recommended by Intel [8], today's operating systems map the kernel into the address space of every user process. Kernel pages are protected from unwanted access by user space applications using different access permissions, set in the page table entries (PTE). Thus, the address space is shared between the kernel and the user and only the privilege level is escalated to execute system calls and interrupt routines.

The idea of *Stronger Kernel Isolation* proposed by Gruss et al. [5] (cf.

[3]Kernel Address Isolation to have Side channels Efficiently Removed.

Figure 10.2) is to unmap kernel pages while the user process is in user space and switch to a separated kernel address space when entering the kernel. Consequently, user pages are not mapped in kernel space and only a minimal numbers of pages is mapped both in user space and kernel space. While this would prevent all microarchitectural attacks on kernel address space information on recent systems [5, 7, 10], it is not possible to implement *Stronger Kernel Isolation* without rewriting large parts of today's kernels. There is no previous work investigating the requirements real hardware poses to implement kernel address isolation in practice. We identified the following three challenges that make kernel address isolation non-trivial to implement.

Challenge 1. Threads cannot use the same page table structures in user space and kernel space without a huge synchronization overhead. The reason for this is the highly parallelized nature of today's systems. If a thread modifies page table structures upon a context switch, it influences all concurrent threads of the same process. Furthermore, the mapping changes for all threads, even if they are currently in the user space.

Challenge 2. Current x86 processors require several locations to be valid for both user space and kernel space during context switches. These locations are hard to identify in large operating system kernels due to implicit assumptions about the omnipresence of the entire kernel address space. Furthermore, segmented memory accesses like core-local storage are required during context switches. Thus, it must be possible to locate and restore the segmented areas without re-mapping the unmapped parts of the kernel space. Especially, unmapping the user space in the Linux kernel space, as proposed by Gruss et al. [5], would require rewriting large parts of the Linux kernel.

Challenge 3. Switching the address space incurs an implicit full TLB flush and modifying the address space causes a partial TLB flush [8]. As current operating systems are highly optimized to reduce the amount of implicit TLB flushes, a countermeasure would need to explicitly flush the TLB upon every context switch. Jang et al. [10] suspected that this may have a severe performance impact.

3.2. Practical Kernel Address Isolation

In this section we show how *KAISER* overcomes these challenges and thus fully revives KASLR.

Shadow Address Spaces. To solve challenge 1, we introduce the idea of *shadow address spaces* to provide kernel address isolation. Figure 10.3 illustrates the principle of the shadow address space technique. Every process has two address spaces. One address space which has the user space mapped but not the kernel (*i.e.*, the *shadow address space*), and a second address space which has the kernel mapped but the user space protected with SMAP and SMEP.

The switch between the user address space and the kernel address space now requires updating the `CR3` register with the value of the corresponding PML4. Upon a context switch, the `CR3` register initially remains at the old value, mapping the user address space. At this point *KAISER* can only perform a very limited amount of computations, operating on a minimal set of registers and accessing only parts of the kernel that are mapped both in kernel and user space. As interrupts can be triggered from both user and kernel space, interrupt sources can be both environments and it is not generally possible to determine the interrupt source within the limited amount of computations we can perform at this point. Consequently, switching the `CR3` register must be a short static computation oblivious to the interrupt source.

With shadow address spaces we provide a solution to this problem. Shadow address spaces are required to have a globally fixed power-of-two offset between the kernel PML4 and the shadow PML4. This allows switching to the kernel PML4 or the shadow PML4 respectively, regardless of the interrupt source. For instance, setting the corresponding address bit to zero switches to the kernel PML4 and setting it to one switches to the shadow PML4. The easiest offset to implement is to use bit 12 of the physical address. That is, the PML4 for the kernel space and shadow PML4 are allocated as an 8 kB-aligned physical memory block. The shadow PML4 is always located at the offset +4 kB. With this trick, we do not need to perform any memory lookups and only need a single scratch register to switch address spaces.

The memory overhead introduced through shadow address spaces is very small. We have an overhead of 8 kB of physical memory per user thread

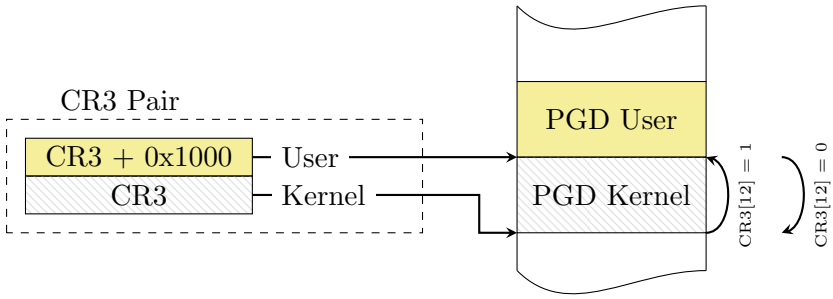


Figure 10.3.: Shadow address space: PML4 of user address space and kernel address space are placed next to each other in physical memory. This allows to switch between both mappings by applying a bit mask to the `CR3` register.

for kernel page directorys (PDs) and PTs and 12 kB of physical memory per user process for the shadow PML4. The 12 kB are due to a restriction in the Linux kernel that only allows to allocate blocks containing 2^n pages. Additionally, *KAISER* has a system-wide total overhead of 1 MB to allocate 256 global kernel page directory pointer tables (PDPTs) that are mapped in the kernel region of the shadow address spaces.

Minimizing the Kernel Address Space Mapping. To solve challenge 2, we identified the memory regions that need to be mapped for both user space and kernel space, *i.e.*, the absolute minimum number of pages to be compatible with x86 and its features used in the Linux kernel. While previous work [5] suggested that only a negligible portion of the interrupt dispatcher code needs to be mapped in both address spaces, in practice more locations are required.

As x86 and Linux are built around using interrupts for context switches, it is necessary to map the interrupt descriptor table (IDT), as well as the interrupt entry and exit `.text` section. To enable multi-threaded applications to run on different cores, it is necessary to identify per-CPU memory regions and map them into the shadow address space. *KAISER* maps the entire per-CPU section including the interrupt request (IRQ) stack and vector, the global descriptor table (GDT), and the task state segment (TSS). Furthermore, while switching to privileged mode, the CPU implicitly pushes some registers onto the current kernel stack. This can be one of the per-CPU stacks that we already mapped or a thread

stack. Consequently, thread stacks need to be mapped too.

We found that the idea to unmap the user space entirely in kernel space is not practical. The design of modern operating system kernels is based upon the capability of accessing user space addresses from kernel mode. Furthermore, SMEP protects against executing user space code in kernel mode. Any memory location that is user-accessible cannot be executed by the kernel. SMAP protects against invalid user memory references in kernel mode. Consequently, the effective user memory mapping is non-executable and not directly accessible in kernel mode.

Efficient and Secure TLB Management. The Linux kernel generally tries to minimize the number of implicit TLB flushes. For instance when switching between kernel and user mode, the `CR3` register is not updated. Furthermore, the Linux kernel uses PTE global bits to preserve mappings that exist in every process to improve the performance of context switches. The global bit of a PTE marks pages to be excluded from implicit TLB flushes. Thus, they reduce the impact of implicit TLB flushes when modifying the `CR3` register.

To solve challenge 3, we investigate the effects of these global bits. We found that it is necessary to either perform an explicit full TLB flush, or disable the global bits to eliminate the leakage completely. Surprisingly, we found the performance impact of disabling global bits to be entirely negligible.

Disabling global bits alone does not eliminate any leakage, but it is a necessary building block. The main side-channel defense in *KAISER* is based on the separate shadow address spaces we described above. As the two address spaces have different `CR3` register values, *KAISER* requires a `CR3` update upon every context switch. The defined behavior of current Intel x86 processors is to perform implicit TLB flushes upon every `CR3` update. Venkatasubramanian et al. [25] described that beyond this architecturally defined behavior, the CPU may implement further optimizations as long as the observed effect does not change. They discussed an optimized implementation which tags the TLB entries with the `CR3` register to avoid frequent TLB flushes due to switches between processes or between user mode and kernel mode. As we show in the following section, our evaluation suggests that current Intel x86 processors have such optimizations already implemented. *KAISER* benefits from these optimizations implicitly and consequently, its TLB management is efficient.

4. Evaluation

We evaluate and discuss the efficacy and performance of *KAISER* on a desktop computer with an Intel Core i7-6700K Skylake CPU and 16GB RAM. To evaluate the effectiveness of *KAISER*, we perform all three microarchitectural attacks applicable to Skylake CPUs (cf. Section 2). We perform each attack with and without *KAISER* enabled and show that *KAISER* can mitigate all of them. For the performance evaluation, we compare various benchmark suites with and without *KAISER* and observe a negligible performance overhead of only 0.08 % to 0.68 %.

4.1. Evaluation of Microarchitectural Attacks

Double Page Fault Attack. As described in Section 2, the double page fault attack by Hund et al. [7] exploits the fact that the page translation caches store information to valid kernel addresses, resulting in timing differences. As *KAISER* does not map the kernel address space, kernel addresses are never valid in user space and thus, are never cached in user mode. Figure 10.4 shows the average execution time of the second page fault. For the default kernel, the execution time of the second page fault is 12 282 cycles for a mapped address and 12 307 cycles for an unmapped address. When running the kernel with *KAISER*, the access time is 14 621 in both cases. Thus, the leakage is successfully eliminated.

Note that the observed overhead for the page fault execution does not reflect the actual performance penalty of *KAISER*. The page faults triggered for this attack are never valid and thus can never result in a valid page mapping. They are commonly referred to as segmentation faults, typically terminating the user program.

Intel TSX-based Attack. The Intel TSX-based attack presented by Jang et al. [10] (cf. Section 2) exploits the same timing difference as the double page fault attack. However, with Intel TSX the page fault handler is not invoked, resulting in a significantly faster and more stable attack. As the basic underlying principle is equivalent to the double page fault attack, *KAISER* successfully prevents this attack as well. Figure 10.5 shows the execution time of a TSX transaction for unmapped pages, non-executable mapped pages, and executable mapped pages. With the default kernel, the transaction execution time is 299 cycles for unmapped pages, 270 cycles for non-executable mapped pages, and 226 cycles for

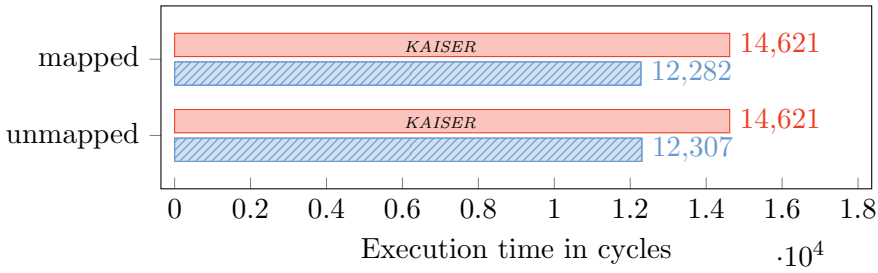


Figure 10.4.: Double page fault attack with and without *KAISER*: mapped and unmapped pages cannot be distinguished if *KAISER* is in place.

executable mapped pages. With *KAISER*, we measure a constant timing of 300 cycles. As in the double page fault attack, *KAISER* successfully eliminates the timing side channel.

We also verified this result by running the attack demo by Jang et al. [9]. On the default kernel, the attack recovers page mappings with a 100 % accuracy. With *KAISER*, the attack does not even detect a single mapped page and consequently no modules.

Prefetch Side-Channel Attack. As described in Section 2, prefetch side-channel attacks exploit timing differences in software prefetch instructions to obtain address information. We evaluate the efficacy of *KAISER* against the two prefetch side-channel attacks presented by Gruss et al. [5].

Figure 10.6 shows the median execution time of the `prefetch` instruction in cycles compared to the actual address translation level. We observed an execution time of 241 cycles on our test system for page translations terminating at PDPT level and PD level respectively. We observed an execution time of 237 cycles when the page translation terminates at the PT level. Finally, we observed a distinct execution times of 212 when the page is present and cached, and 515 when the page is present but not cached. As in the previous attack, *KAISER* successfully eliminates any timing differences. The measured execution time is 241 cycles in all cases.

Figure 10.7 shows the address-translation attack. While the correct guess

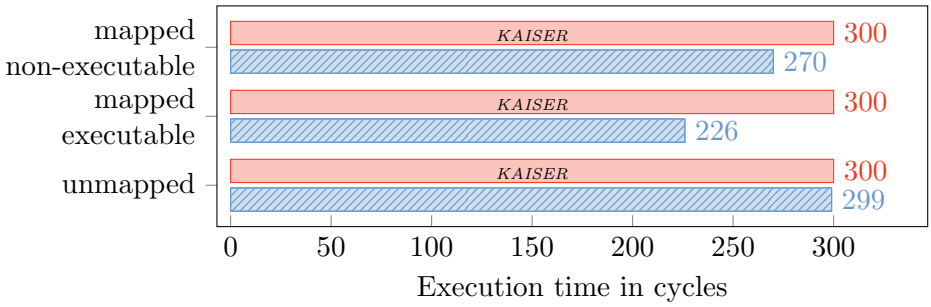


Figure 10.5.: Intel TSX-based attack: On the default kernel, the status of a page can be determined using the TSX-based timing side channel. *KAISER* completely eliminates the timing side channel, resulting in an identical execution time independent of the status.

can clearly be detected without the countermeasure (dotted line), *KAISER* eliminates the timing difference. Thus, the attacker is not able to determine the correct virtual-to-physical translation anymore.

4.2. Performance Evaluation

As described in Section 3.2, *KAISER* has a low memory overhead of 8 kB per user thread, 12 kB per user process, and a system-wide total overhead of 1 MB. A full-blown Ubuntu Linux already consumes several hundred megabytes of memory. Hence, in our evaluation the memory overhead introduced by *KAISER* was hardly observable.

In order to evaluate the runtime performance impact of *KAISER*, we execute different benchmarks with and without the countermeasure. We use the PARSEC 3.0 [1] (input set "native"), the Kaiserpgbench [24] and the SPLASH-2x [16] (input set "native") benchmark suites to exhaustively measure the performance overhead of *KAISER* in various different scenarios.

The results of the different benchmarks are summarized in Figure 10.8 and Table 10.1. We observed a very small average overhead of 0.28% for all benchmark suites and a maximum overhead of 0.68% for single tests. This surprisingly low performance overhead underlines that *KAISER* should be deployed in practice.

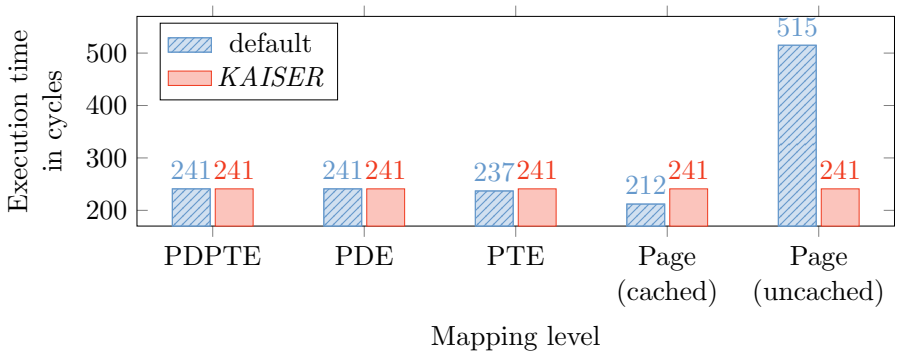


Figure 10.6.: Median prefetch execution time in cycles depending on the level where the address translation terminates. With the default kernel, the execution time leaks information on the translation level. With *KAISER*, the execution time is identical and thus does not leak any information.

Table 10.1.: Average performance overhead of *KAISER*.

Benchmark	Kernel	Runtime				Average Overhead
		1 core	*2 cores*	*4 cores*	*8 cores*	
PARSEC 3.0	default	27:56,0 s	14:56,3 s	8:35,6 s	7:05,1 s	0.37 %
	KAISER	28:00,2 s	14:58,9 s	8:36,9 s	7:08,0 s	
pgbench	default	3:22,3 s	3:21,9 s	3:21,7 s	3:53,5 s	0.39 %
	KAISER	3:23,4 s	3:22,5 s	3:22,3 s	3:54,7 s	
SPLASH-2X	default	17:38,4 s	10:47,7 s	7:10,4 s	6:05,3 s	0.09 %
	KAISER	17:42,6 s	10:48,5 s	7:10,8 s	6:05,7 s	

4.3. Reproducibility of Results

In order to make our evaluation of efficacy and performance of *KAISER* easily reproducible, we provide the source code and precompiled Debian packages compatible with Ubuntu 16.10 on GitHub. The repository can be found at `https://github.com/IAIK/KAISER`. We fully document how to build the Ubuntu Linux kernel with *KAISER* protections from the source code and how to obtain the benchmark suites we used in this evaluation.

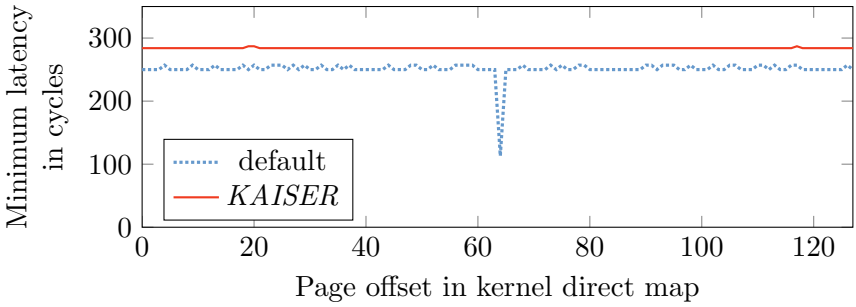


Figure 10.7.: Minimum access time after prefetching physical direct-map addresses. The low peak in the dotted line reveals to which physical address a virtual address maps (running the default kernel). The solid line shows the same attack on a kernel with *KAISER* active. *KAISER* successfully eliminates the leakage.

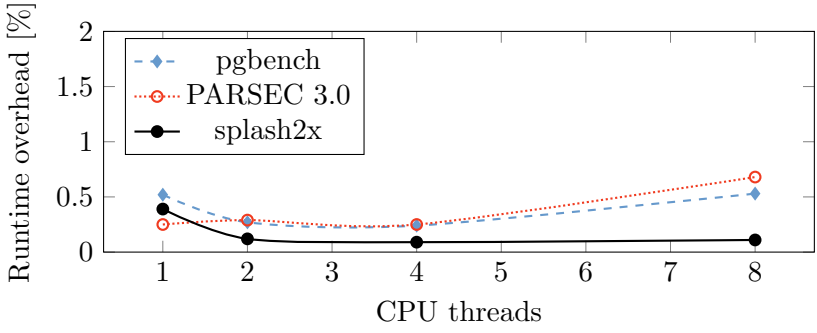


Figure 10.8.: Comparison of the runtime of different benchmarks when running on the *KAISER*-protected kernel. The default kernel serves as baseline (=100%). We see that the average overhead is 0.28% and the maximum overhead is 0.68%.

5. Future Work

KAISER does not consider BTB attacks, as they require knowledge of the BTB behavior. The BTB behavior has not yet been reverse-engineered for recent Intel processors, such as the Skylake microarchitecture (cf. Section 2.3). However, if the BTB is reverse-engineered in future work, attacks on systems protected by *KAISER* would be possible. Evtyushkin et al. [3] proposed to use virtual address bits > 30 to randomize memory locations for KASLR as a zero-overhead countermeasure against BTB attacks. *KAISER* could incorporate this adaption to effectively mitigate BTB attacks as well.

Intel x86-64 processors implement multiple features to improve the performance of address space switches. Linux currently does not make use of all features, e.g., Linux could use process-context identifiers to avoid some TLB flushes. The performance of *KAISER* would also benefit from these features, as *KAISER* increases the number of address space switches. Consequently, utilizing these optimization features could lower the runtime overhead below 0.28%.

KAISER exploits very recent processor features which are not present on older machines. Hence, we expect higher overheads on older machines if *KAISER* is employed for security reasons. The current proof-of-concept implementation of *KAISER* shows that defending against the attack is possible. However, it does not eliminate all KASLR information leaks, especially information leaks that are not caused by the same hardware effects. A full implementation of *KAISER* must map any randomized memory locations that are used during the context switch at fixed offsets. This is straightforward, as we have already introduced new mappings which can easily be extended. During the context switch, kernel memory locations are only accessed through these fixed mappings. Hence, the offsets of the randomized parts of the kernel can not be leaked in this case.

6. Conclusion

In this paper we discussed limitations of x86 impeding practical kernel address isolation. We show that our countermeasure (*KAISER*) overcomes these limitations and eliminates all microarchitectural side-channel attacks on kernel address information on recent Intel Skylake systems. More specifically, we show that *KAISER* protects the kernel against double page

fault attacks, prefetch side-channel attacks, and TSX-based side-channel attacks. *KAISER* enforces a strict kernel and user space isolation such that the hardware does not hold any information about kernel addresses while running user processes. Our proof-of-concept is implemented on top of a full-fledged Ubuntu Linux kernel. *KAISER* has a low memory overhead of approximately 8 kB per user thread and a low runtime overhead of only 0.28%.

Acknowledgments

We would like to thank our anonymous reviewers, Anders Fogh, Rodrigo Branco, Richard Weinbeger, Thomas Garnier, David Gens and Mark Rutland for their valuable feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

References

[1] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[2] Rodrigo Branco and Shay Gueron. "Blinded random corruption attacks". In: *IEEE International Symposium on Hardware Oriented Security and Trust (HOST'16)*. 2016.

[3] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR". In: *International Symposium on Microarchitecture (MICRO'16)*. 2016.

[4] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU". In: *NDSS'17*. 2017.

[5] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". In: *CCS'16*. 2016.

[6] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *DIMVA'16*. 2016.

[7] Ralf Hund, Carsten Willems, and Thorsten Holz. “Practical Timing Side Channel Attacks against Kernel Space ASLR”. In: *S&P’13*. 2013.

[8] Intel. “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3 (3A, 3B & 3C): System Programming Guide”. In: 253665 (2014).

[9] Yeongjin Jang. *The DrK Attack - Proof of concept*. `https://github.com/sslab-gatech/DrK`. Retrieved on February 24, 2017. 2016.

[10] Yeongjin Jang, Sangho Lee, and Taesoo Kim. “Breaking Kernel Address Space Layout Randomization with Intel TSX”. In: *CCS’16*. 2016.

[11] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. “ret2dir: Rethinking kernel isolation”. In: *USENIX Security Symposium*. 2014, pp. 957–972.

[12] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. In: *ISCA’14*. 2014.

[13] Kirill A. Shutemov. *Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace*. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce`. Retrieved on November 10, 2015. Mar. 2015.

[14] Jonathan Levin. *Mac OS X and IOS Internals: To the Apple’s Core*. John Wiley & Sons, 2012.

[15] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: *NDSS’17*. to appear. 2017.

[16] PARSEC Group. *A Memo on Exploration of SPLASH-2 Input Sets*. `http://parsec.cs.princeton.edu`. 2011.

[17] PaX Team. *Address space layout randomization (ASLR)*. `http://pax.grsecurity.net/docs/aslr.txt`. 2003.

[18] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security Symposium*. 2016.

[19] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows internals*. Pearson Education, 2012.

[20] Mark Seaborn and Thomas Dullien. “Exploiting the DRAM rowhammer bug to gain kernel privileges”. In: *Black Hat 2015 Briefings*. 2015.

[21] Hovav Shacham. “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)”. In: *14th ACM CCS*. 2007.

[22] Hovav Shacham, Matthew Page, Ben Pfaff, EuJin Goh, Nagendra Modadugu, and Dan Boneh. “On the effectiveness of address-space randomization”. In: *CCS'04*. 2004.

[23] Solar Designer. *Getting around non-executable stack (and fix)*. `http://seclists.org/bugtraq/1997/Aug/63`. Aug. 1997.

[24] The PostgreSQL Global Development Group. *pgbench*. `https://www.postgresql.org/docs/9.6/static/pgbench.html`. 2016.

[25] Girish Venkatasubramanian, Renato J. Figueiredo, Ramesh Illikkal, and Donald Newell. “TMT: A TLB Tag Management Framework for Virtualized Platforms”. In: *International Journal of Parallel Programming* 40.3 (2012).

EXPLOITING MICROARCHITECTURAL OPTIMIZATIONS FROM SOFTWARE

MORITZ LIPP PHD THESIS

With abstraction layers, the implementation details of software and hardware components are hidden away to deal with the complexity of modern computer systems. While the Instruction Set Architecture (ISA) serves as an interface between the CPU and the software running on it, the computer microarchitecture is the actual hardware implementation of the ISA. The clearly defined interfaces do not only cover up the complexity but also allow different variants of the microarchitecture to be built. While they all fulfill the contract defined by the ISA, they can differ in other aspects, such as performance, security, energy efficiency, or other physical properties. Microarchitectural attacks exploit these variations occurring on the microarchitectural level of modern CPUs. With side-channel attacks and fault attacks, there are different ways that allow learning from and tampering with the actual implementation. These attacks allow adversaries to extract sensitive information processed on the system, e.g., cryptographic keys or user behavior.

In this thesis, we expand the landscape of software-based microarchitectural attacks and defenses. By exploring the security implications of different optimizations, we identify previously unknown attack vectors, allowing us to circumvent the most fundamental security guarantees of modern processors. We combine traditional physical side-channel analyses with software-based microarchitectural attack techniques to leak sensitive information processed on the CPU. We enlarge our understanding of which settings and circumstances facilitate different existing attacks and give new insights into developing effective and efficient mitigations.

TU Graz

Graz University of Technology
Faculty of Computer Science
Institute for Applied Information Procesing and Communications