Revisiting Isolated and Trusted Execution via Microarchitectural Cryptanalysis

by

Ahmad "Daniel" Moghimi

A Dissertation Submitted to the Faculty

of the

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Electrical and Computer Engineering

Abstract

Shared computing resources shaping modern computing and the internet ecosystem introduce new security and privacy challenges. For instance, in a virtualized environment like the *cloud*, multiple users with virtually isolated security domains share the CPU and system memory. A malicious user may exploit microarchitectural side channels like the cache timing to snoop on other users' memory access patterns in this environment. Such memory snooping attacks are also possible in other shared execution environments such as web browsers and smartphones. As a result of these attacks, security-sensitive applications, e.g., cryptographic protocols, require extra care against the danger of leaking secret bits to adversaries. Additionally, some attacks like the *rowhammer* go beyond compromising *confidentiality*. On a system with shared memory, rowhammer can compromise the *integrity* of applications by intentionally inducing memory errors.

Microarchitectural side channels are severe threats to security and privacy concerning the growth of multitenancy, Consequently, researchers have recently proposed several mitigations to circumvent these attacks. However, these mitigations, for the most part, are based on the limited understanding of the microarchitecture and potential attack vectors. As some of our contributions highlight, we can construct new information channels based on low-level analysis and micro-benchmarking of the CPU's memory subsystem. Based on our findings, we propose multiple contention-based techniques that improve previous attack vectors. By looking at the memory subsystem with more scrutiny, we show that existing mitigations against memory-related side-channel leakage are insufficient.

The complex microarchitecture also exposes the software layer to a new class of attacks, *transient execution attacks*. In contrast to the aforementioned contention-based attacks, microarchitectural data sampling (MDS) allow a local adversary to leak the actual data bits rather than memory access patterns. Therefore, attackers will have full visibility to steal credentials and data from other users who run on the same CPU core. However, manual analysis and testing of some transient execution attacks like the MDS do not scale and limit our understanding of these vulnerabilities' root causes. To automate sophisticated proof of concepts and find new variants, we developed a tool by adopting software vulnerability fuzzing techniques. With our automated approach, we provide new insights, discover new exploitation techniques, and report new vulnerabilities.

Microarchitectural vulnerabilities go beyond affecting traditional software and security boundaries. A prominent element of modern processors and shared computing environments are the support for hardware-based trusted computing. For example, trusted execution environments (TEEs) are now available on various processors, including superscalar CPUs, mobile processors, and embedded systems. TEEs promise a wide range of security and privacy applications, such as privacy-preserving artificial intelligence and digital right management. However, TEEs face a more challenging threat model, especially for microarchitectural security, as the system software, including the operating system, is considered malicious. While it is intuitive that TEEs are as vulnerable to microarchitectural attacks, we present that the unique adversarial model suggested by a TEE like the Intel SGX exposes the trusted computation to unusual and innovative attack vectors. We show that an adversarial operating system can exploit its system-level capabilities and architectural features to leak fine-grained and deterministic side-channel information from secure *enclaves*, which are not possible in traditional threat models.

TEEs are not the only relevant hardware-based trusted computing solution. cryptographic co-processors like the *trusted platform module (TPM)* are responsible for executing cryptographic operations in a physically-isolated fashion. TPMs even promise security guarantees against more intrusive side-channel attacks like physical probing and tampering. While TPM devices claim such security guarantees through external evaluation and security certification, we show that the obscurity of these cryptographic co-processors leaves them vulnerable to classic timing attacks. As a result, we develop high-precision timers to perform timing analysis of cryptographic operations inside TPMs empirically.

Conclusively, to show the impact of security failures due to the above software-related side-channel and microarchitectural attacks, we demonstrate several realistic end-to-end attacks. In particular, cryptographic protocols are an essential ingredient of security primitives for network security, secure software isolation, and trusted execution environments. By combining the newly discovered attack vectors with theoretical cryptanalysis techniques and devising new algorithmic approaches, we demonstrate practical attacks to steal secret keys from encryption and digital signature operations. Our findings include discovering several critical vulnerabilities on deployed cryptographic products ranging from standard cryptographic libraries to hardware-based security solutions.

In retrospect, we present the ideas, tools, and techniques under the framework of microarchitectural cryptanalysis. This framework helps the community to have a better understanding of security issues concerning complex microarchitectures. We discuss the importance of applying microarchitectural cryptanalysis to future systems having a heterogeneous microarchitecture. Microarchitectural cryptanalysis highlights the essential

need for developing analysis and automation tools in this direction. We hope that our contribution will help the reader rethink threat models, design choices, and engineering practices for secure systems development.

Acknowledgements

My deepest gratitude goes to my dissertation advisor, Professor Berk Sunar, for his trust and full support throughout my years of Ph.D. studies. He gave me lots of freedom to pursue new and exciting research ideas, and it was always a pleasure to work with him on several projects closely. I would also like to thank Professor Thomas Eisenbarth, who was initially my primary advisor at the ECE department of WPI. After he moved to pursue a new role at the University of Luebeck, we continued to have successful collaborations. I also want to thank my external committee member, Professor Simha Sethumadhavan from Columbia University, for his time, attention, and feedback on this dissertation. I would also like to acknowledge my colleagues, Berk, Saad, and Zane from WPI, Thore, Ida, and Jan from the University of Luebeck for great discussions during several projects.

Next, I would like to acknowledge the talented researchers from other institutions whom I was honored to work with during several projects: I learned a lot from and enjoyed working with Moritz Lipp and Michael Schwarz from TU Graz, and Jo Van Bulck from KU Leuven. Professor Daniel Gruss from TU Graz and Professor Frank Piessens from KU Leuven were tremendously encouraging, supportive, and communicative during several collaborations. I learned more about cryptanalysis from Professor Nadia Heninger, who hosted me as a visiting graduate student at the University of California, San Diego. She immensely helped me to improve the quality of my publications.

The work in this dissertation was supported by the US National Science Foundation under grants no. 1618837, and 1814406. Additional funds were provided by research gifts from Intel and Cloudflare.

Publications

The majority of materials produced as part of this work have been published earlier in peer-reviewed conference proceedings and journals, including:

- the *Usenix Security Symposium* [187, 247, 248, 248],
- the *ACM Computer and Communications Security* [61, 300],
- the *IEEE Security & Privacy* [349],
- the *IACR Conference and Transactions on Cryptographic Hardware and Embedded Systems* [82, 245, 367],
- the *Cryptographers' Track at the RSA Conference* [244],
- the *International Journal of Parallel Programming* [246],
- the *Annual Computer Security Applications Conference* [368].

As the main contributor, I have incorporated the following publications directly with minor editorials to shape this dissertation's content.

1. **D Moghimi**, J Van Bulck, N Heninger, F Piessens, B Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves" The 29th USENIX Security Symposium. 2020.

2. **D Moghimi**, M Lipp, B Sunar, M Schwarz. "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis" The 29th USENIX Security Symposium. 2020.

3. **D Moghimi**, B Sunar, T Eisenbarth, N Heninger. "TPM-Fail: TPM meets Timing and Lattice Attacks" The 29th USENIX Security Symposium. 2020.

4. S Islam, **A Moghimi**, I Bruhns, M Krebbel, B Gulmezoglu, T Eisenbarth, B Sunar. "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks" The 28th USENIX Security Symposium. 2019.

5. **A Moghimi**, J Wichelmann, T Eisenbarth, B Sunar. "MemJam: A False Dependency Attack against Constant-Time Crypto Implementations" (Extended Version) International Journal of Parallel Programming 47.4 (2019).

6. **A Moghimi**, T Eisenbarth, B Sunar. "MemJam: A False Dependency Attack against Constant-Time Crypto Implementations in SGX" Cryptographers' Track at the RSA Conference. 2018.

I have excluded the following publications from this dissertation. We mention them when it is appropriate as part of our related contributions.

7. Z Weissman, T Tiemann, **D Moghimi**, E Custodio, T Eisenbarth, B Sunar. "JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms" IACR Transactions on Cryptographic Hardware and Embedded Systems. 2020, 3 (Jun. 2020).

8. J Van Bulck, **D Moghimi**, M Schwarz, M Lipp, M Minkin, D Genkin, Y Yarom, B Sunar, D Gruss, F Piessens."LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection" The 41st IEEE Symposium on Security and Privacy. 2020.

9. M Schwarz, M Lipp, **D Moghimi**, J Van Bulck, J Stecklina, T Prescher, D Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling" Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019.

10. M Minkin, **D Moghimi**, M Lipp, M Schwarz, J Van Bulck, D Genkin, D Gruss, F Piessens, B Sunar, Y Yarom. "Fallout: Reading Kernel Writes From User Space" arXiv preprint arXiv:1905.12701 merged with Store-to-Leak (arXiv:1905.05725), under Canella et al. "Fallout: Leaking Data on Meltdown-resistant CPUs" Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019.

11. J Wichelmann, **A Moghimi**, T Eisenbarth, B Sunar. "MicroWalk: A Framework for Finding Side Channels in Binaries" Proceedings of the 34th Annual Computer Security Applications Conference. 2018.

12. F Dall, G De Micheli, T Eisenbarth, D Genkin, N Heninger, **A Moghimi**, Y Yarom. "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks" IACR Transactions on Cryptographic Hardware and Embedded Systems. 2018, 2 (May 2018).

13. **A Moghimi**, G Irazoqui, T Eisenbarth. "CacheZoom: How SGX Amplifies The Power of Cache Attacks" International Conference on Cryptographic Hardware and Embedded Systems. Springer, Cham, 2017.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer security professionals and cryptographers have spent decades designing systems with more robust security promises. However, due to the complexity of engineering computers, bug-free and reliable implementation of these designs is challenging. In the past, these challenges have introduced numerous vulnerabilities that could damage the economy, privacy, and reputation of individuals and companies. On the bright side, recent designs and development tools can eliminate the majority of traditional software vulnerabilities or mitigate their risk on a large scale using a combination of hardware and software techniques [4, 221]. While these efforts have partially improved security and privacy, software execution on commodity processors may not be as reliable as perceived.

As we have rapidly filled data centers, workspaces, and homes with extremely efficient processors and have added new features, we have also added more complexity to the hardware design and engineering. As a result, security-critical software and cryptographic implementations are executed based on uncertainties about the underlying hardware that may lead to new classes of vulnerabilities. In this work, we uncover some of the complexity of modern computing hardware with particular attention to security and privacy. More specifically, we focus on the interface between the hardware and software and discuss hardware-related security issues exploitable from the software. These security issues, if exploited, challenge existing security and isolation boundaries. In this chapter, after setting the stage with the preliminaries of security technologies (§1.1), we further motivate the goal of our work (§1.2). Then, we provide an overview of our contributions (§1.3) and lay out the organization of this writing (§1.4).

1.1 Computer Security and Cryptography

Over time, the diversity of emerging threat models has encouraged designing several complimentary security primitives and protocols. In this section, we give a high-level overview of communication security (§1.1.1), secure isolation techniques in multitenant environments (§1.1.2), and trusted computing (§1.1.3). Finally, we discuss the historical pitfalls of engineering and implementations issues that undermine the goal of secure designs (§Section 1.1.4).

1.1.1 Communication Security

Network security protocols. Securing the communication between two separate computers of any kind, e.g., servers, workstations, mobile computers, is essential to defend against network adversaries. Nowadays, it is rare to find a security-critical internet service that does not use one of the standard security protocols for protecting the communication between service providers and clients. Security protocols such as *Transport Layer Security (TLS)* [88], *Internet Key Exchange (IKE)* [144], and *Secure Shell (SSH)* [387] encrypt the transmitted data between a client and a server, or more broadly, the two endpoints over a communication channel. A secure encryption scheme like the *Advanced Encryption Standard (AES)* [81] encrypts the data with a high-entropy key that is not guessable using brute-force attacks, *i.e.*, trying out all the possible key combinations. In theory, since only the client and server have access to this encryption key, a passive *man-in-the-middle (MITM) attacker* who can only read the traffic between the client and the server, can not access the transmitted information in plaintext.

More broadly, security protocols like the TLS provide an end-to-end secure communication channel by also protecting against *active* MITM attackers who try to modify the traffic or impersonate an endpoint inside the network. For this, a typical protocol also supports secure key establishment based on the *Diffie-Hellman (DH)* scheme [280], authentication based on a *public-key cryptographic (PKC)* scheme like the *elliptic curve digital signature algorithm (ECDSA)* [191], and integrity measurement using a reliable cryptographic hash algorithm like the *SHA-3* [95]. In reality, several other cryptographic primitives and protocol-level designs are involved in a protocol like TLS [77]. In a nutshell, these protocols glue several cryptographic schemes to provide a secure end-to-end communication channel with guarantees such as confidentiality, integrity, and authenticity.

Computational security and cryptanalysis. The claimed security guarantees of a cryptographic scheme rely on the computational capability of available computers. In this respect, a cryptographic design is considered secure if adversaries can not execute brute-force or any other cryptanalysis attacks efficiently. Without any breakthrough in computing hardware or discovering a severe design flaw in a target scheme, brute-force attacks take millions of years to try all possible key and input combinations required to succeed in an attack. However, in the past, cryptanalysts have discovered several design flaws and vulnerabilities in earlier versions of security protocols and the underlying cryptographic schemes. Some of these design flaws allow adversaries to efficiently undermine the claimed security guarantees, or worse, extract secret keys and information by just observing the input and output of algorithms. For example, earlier versions of TLS used hash functions such as MD5 [283] and SHA1 [96]. Today, due to fundamental design issues, MD5 and SHA1 are considered insecure, as adversaries can trivially find *hash collisions* for these algorithms [325, 361]. Researchers have also demonstrated several other attacks against earlier versions of a protocol like TLS due to design flaws at the protocol level [9, 23, 250, 310]. Thankfully, the community has addressed previous design issues in the latest specification of TLS, and they have deprecated the usage of such insecure cryptographic primitives [279]. While there is a chance for vulnerabilities to be found in the future, we consider state-of-the-art cryptographic designs to have strong security guarantees against traditional cryptanalysis [80].

Can quantum computers break it all? We mentioned that the claimed security of cryptographic schemes also depends on whether we can build computers that are exceptionally capable of executing specific cryptanalysis or brute-force attacks.

Theoreticians have argued that once quantum computers reach a particular capability (maybe in a few decades), they can break the security of some of the cryptographic algorithms that were previously known to be secure by design. For example, the *RSA* algorithm used for encryption and digital signatures relies on the difficulty of factoring large integers. However, *Shore's algorithm* proves that factoring large integers is relatively trivial on a futuristic quantum computer with thousands of qubits [313]. Therefore, RSA will not be secure, and all previous RSA keys are deemed vulnerable once quantum computers reach the estimated capability. Although post-quantum computers with such capability are not accessible today, cryptographers have been preparing for security in a post-quantum world; they have already designed several alternative schemes that will be secure even against a powerful quantum computer [71].

With all these improvements and considerations in cryptography, we can conclude that humans have become significantly good at designing cryptographic protocols and schemes that are incredibly secure today and even in the future. However, in practice, these designs fall short in protecting assets when implemented and executed using software, hardware, or a combination of both. Understanding security vulnerabilities stemming from computer architectures is orthogonal to securely deploying these theoretically secure mathematical abstractions today and in the future.

How things are executed. One can implement cryptographic schemes in software that runs on a general-purpose processor. Alternatively, for a more efficient execution, hardware engineers may develop *application-specific integrated circuit (ASIC)* [317] or logics for *field programmable gate arrays (FPGAs)* [338]. In practice, for most use cases, cryptographic schemes and protocols are only partially implemented in hardware. End-to-end use cases combine software and hardware techniques. The software executes the higher-level and more complex algorithms, and the software interface with specific hardware blocks to do certain low-level logic. For example, nowadays, most general-purpose processors from Intel, AMD, and ARM support specific instructions to facilitate AES encryption [13]. Proprietary hardware logic within these processors efficiently executes these instructions. Despite these instructions' availability, the software is still responsible for implementing the high-level logic of encryption and decryption, programming and managing keys, modes of operations, and padding schemes used in security protocols and applications.

During the design and development of hardware or software for cryptography, engineers may introduce new security vulnerabilities into the system. These vulnerabilities may occur due to various reasons including, but not limited to, human errors [94], invalid assumptions [357], or improper threat modeling [373]. The complexity of modern computing systems provides a suitable environment for such failures. Vulnerabilities that occur during the engineering phase provide a considerable gap between theoretical security guarantees and practical ones. Imagine that a developer mistakenly programs 128-bit encryption keys for AES with a random number generator that only provides 40 bits worth of entropy. A brute-force attack on AES-128 should take up to 2^{128} triage, which is out of reach on any computer, but as soon as attackers find out how the keys are generated, they can narrow down their attempts to 2^{40} triage, which can be computed even on a personal computer.

Although we just outlined such security engineering flaws in the context of cryptographic schemes, any security-critical algorithm may as well suffer from vulnerabilities not seen as part of the abstract design. In general, as we will discuss more extensively in Section 1.1.4, the promised security claimes in abstract or mathematical designs are not always guaranteed when one transforms the abstraction into concrete designs and implementations of hardware or software.

1.1.2 Resource Sharing and Secure Isolation

For a while, computers were operated by a single user to perform a single task. The single-user single-task computing model can not achieve the best performance from the growing number of transistors on processors and the resulting rich computing resources. Consequently, operating systems started to support multi-tasking. Network applications have encouraged multi-user use cases in which different users can log in with their credentials into a single workstation. With the growing computation capability of workstations, rich web content driven by dynamic programming environments like JavaScript has become a reality [252]. Data centers in the cloud environment have become capable of hosting hundreds of processor cores on a single machine used by several service providers. In summary, secure sharing and management of hardware resources among different users on the same machine play essential roles in optimizing the usage of computation resources.

Multiple users with different security and privacy roles execute separate software instances on the same processor and memory subsystem in such multitenant environments. It is intuitive that without a mechanism to isolate applications from accessing each others' memory, a malicious user would steal credentials, cryptographic keys, and other users' information. Modern computing systems consider several isolation boundaries to prevent such compromises. Web browsers have to isolate dynamic content, e.g., JavaScript programs from untrusted origins, to access sensitive information like the session cookies of other origins or systems credentials. Applications such as web and mobile *apps* require to be isolated, as each application has different security roles and permissions. In the cloud environment, service providers must isolate the computing spaces rented by untrusted users, so adversaries can not scan the entire memory to find the *sysadmin* credentials and information from all customers co-located on the same machine.

Secure isolation and local adversaries. Network protocols aim to isolate and protect the transmitted information across separate computers from **network adversaries**. Additionally, with multitenancy on both client- and server-side computers, secure isolation of different applications that run on shared computing resources are necessary to block **local adversaries** and secure network credentials. Hardware and software vendors have worked together to design several architectural support for the isolation of different security domains on the *operating system (OS)*, cloud environments, mobile devices, and web browsers. Some of these isolation techniques are software-only approaches based on programming languages and compiler technologies, e.g., JavaScript. Other hardware-supported techniques isolate different security domains based on hardware-software contracts supported by the processor and managed by the system software, e.g., OS or *hypervisor*.

The *instruction set architecture (ISA)* of the processor provides a hardware-software contract for the modern OS to implement several resource management and security features such as virtual memory manager, task scheduler, assigning roles, and enforcing permissions [330]. Using these features, the OS provides process-level isolation, a virtually isolated environment for each process, which prevents a malicious process from reading or modifying other processes' memory content. Processes can also be assigned to different roles. Therefore, a user-level process will only have access to a limited set of resources, and it can not modify system-level resources to subvert process- and user-level isolations. However, if malicious users find and exploit a vulnerability in the OS kernel, driver software, other privileged processes, or the hardware, they can escalate their privilege [70].

Similar to process-level isolation, the ISA supports other features like the *extended page tables* to allow full-system virtualization and isolation. In this scenario, a hypervisor software manages physical resources, and each virtual machine (VM) executes a full-featured guest OS, e.g., Linux, Microsoft Windows. By design, the guest OS should not have access to the file system, memory, and in general data of other guest OSs or the hypervisor, *i.e.*, VM-level isolation. However, if adversaries find vulnerabilities inside the hypervisor software or the hardware, they have a chance to compromise the entire system [272, 366].

Programming languages can enable another form of architectural isolation by defining a new architecture on top of the native ISA. For example, a widespread use case of such software-based isolation is to sandbox programs written in JavaScript or web assembly (Wasm) [139]. This scenario is useful when a service provider or a malicious endpoint provides the untrusted code. The runtime environment executes the untrusted code within the same virtual address space as the browser's process. However, the runtime environment provides architectural guarantees to contain the untrusted code to access well-defined virtual address space sections.

As we extensively discuss in this dissertation, architectural guarantees do not always translate to strong isolation due to processors' microarchitecture. For example, a microarchitectural attack like the Spectre [207] can compromise the confidentiality guarantee of all of these isolation techniques, as it allows microarchitectural data to become visible at the architecture level. Therefore, it is essential to understand the severity of hardware-based vulnerabilities that can be exploited by the software. Unlike software vulnerabilities, a single microarchitectural vulnerability may allow subversion of several forms of isolation boundaries [127].

1.1.3 Trusted Computing

Even if we assume that all the security protocols and isolation techniques described earlier are going to work flawlessly, we still have to trust many entities to store and process private data such as financial information, medical data, and personal contacts. A single computer on the network includes software components and hardware parts from tens of different manufacturers. Users have to trust these entities as they voluntarily give up their data ownership as soon as they start using the network and computing infrastructure. In the current internet ecosystem, we assume that these manufacturers are neither malicious nor hackable, which the latter is a big stretch considering recent data breaches [60, 124].

In an ideal world, we would like users to maintain ownership of their data, while computers should only serve users and process data without accessing it in plaintext. To achieve this goal, cryptographers have designed several mathematical solutions under the umbrella of *fully-homomorphic encryption (FHE)* [118]. FHE allows computers to compute with encrypted data in a unique form without enabling the data to be revealed. Each user encrypts their data with their secret FHE key that transforms the data into this particular encrypted form. Later on, the user can recover the result of this encrypted computation by decrypting the encrypted results. As a result, with FHE, the user only has to trust a single workstation owned by the user to encrypt the data and decrypt the encrypted output.

Hardware-based trusted computing. Trusted computing generally refers to allowing users to take ownership of their data by redefining trust boundaries across computing systems. Although FHE can theoretically promise this ideal notion, it is not practical, as it requires a tremendous amount of computation and memory. Consequently, deployed trusted computing solutions trade this idealism with efficiency and practicality by relying on hardware-based trusted computing technologies. In this case, the user relies on specific hardware features and certain assumptions about the threat model to partially achieve trusted computation. For example, a *cryptographic co-processor* allows the user to trust a very restricted chip connected to the machine to perform critical operations like cryptographic transactions. Users of this chip still require to trust the manufacturer to perform trusted encryption and authentication, but they distrust the computation and software running on the *central processing unit (CPU)*. The manufacturers also have to make particular assumptions about the adversarial model for developing this chip, e.g., *Can attackers physically access the bus between the chip and the memory, decap the chip, or induce glitches?* Assuming that these assumptions have not underestimated the attacker, the security chip provides a far from the ideal but usable notion of trusted computing.

A side benefit of hardware-based trusted computing is to reduce the attack surface, as security-critical information will be contained to a small subset of the system. Reducing the attack surface is helpful since even without malicious intent, implementation vulnerabilities, which compromise various components, can not access all the critical data. However, we will see that these hardware-based trusted techniques may themselves suffer from improper threat modeling and vulnerabilities.

Cryptographic co-processors generally provide strong isolation, *i.e.*, physical separation at the chip level. As a result, even if the entire CPU is compromised, attackers can not access cryptographic keys stored within the chip. Like the Trusted Platform Module (TPM) [342], some of these co-processors even promise security guarantees against physical adversaries. The TPM standard defines a set of cryptographic primitives known to be secure. These products also need to go under third-party security evaluation to meet Common Criteria (CC) certification [341]. Such certifications should theoretically improve the security of such products when it comes to implementation security. However, we will see that this assurance is only as sound as the quality of tests, which is generally a vague and proprietary process.

Trusted execution environments. While the physical separation of cryptographic co-processors promises strong guarantees, they are only limited to execute a limited set of cryptographic operations. They are also not suitable for executing computationally-intensive operations, as they are generally developed based on low-powered embedded processing technologies. Allocating high-performance computing resources solely to perform a limited number of operations is not an attractive option for processor manufacturers dedicated to achieving a better performance.

Hardware-based techniques known as the trusted execution environment (TEE) rely on the CPU architecture to draw new security boundaries. Manufacturers generally design and implement TEEs to mitigate against system-level adversaries such as the OS, hypervisor, or the BIOS software. Some TEEs like Intel SGX even promises isolation against physical adversaries who try to access the memory bus [132, 174]. With a TEE like Intel SGX, users can execute arbitrary computations inside architecturally isolated regions of the memory and CPU without allowing the OS to access these regions at runtime. As a result, manufacturers have repurposed these high-performance computing resources for both trusted and regular computing.

The benefit of TEEs is that users can execute trusted applications efficiently and with flexibility. Therefore TEEs have been used for a wide range of applications, including copy protection [28], confidential computing [240], privacy-preserving machine learning [274], and private blockchain transaction [46]. On the downside, with TEEs, security engineers now have to defend against new software adversaries who have much more capability than network or local adversaries. Microarchitectural attacks conducted by an adversarial operating system is a new avenue of research that we have started looking into in 2016 [245, 347]. As we show in some of our contributions, a **system-level adversary** is more successful in performing specific side-channel and microarchitectural attacks against the trusted applications running on the same CPU. These attacks show that demonstrated

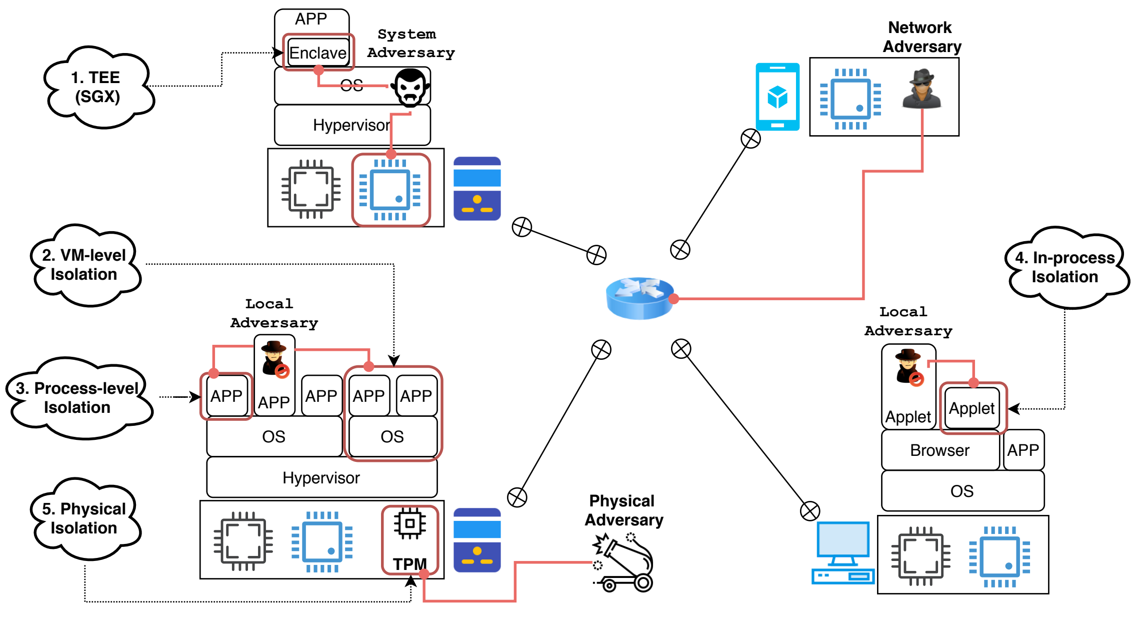


Figure 1.1: A traditional network adversary (top-right) targets the communication between different computers. However, on the endpoints, we see several other threat models. For a TEE (1), a compromised OS is a powerful adversary trying to steal information from a secure module (enclave). In other isolated software environments (2, 3, 4), we see a local adversary with minor privilege trying to compromise other apps, processes, or VMs. For a physically-isolated device like TPM (5), a physical adversary is free to perform the most intrusive attacks.

use cases of TEEs with overly optimistic performance metrics may provide a false sense of security.

Figure 1.1 illustrates several different threat models that we have mentioned.

1.1.4 Pitfalls of Security Engineering

Abstract designs of the above security technologies have not always been immune to vulnerabilities. Still, our community has established a better understanding of threat modeling and secure design at this stage over the years. However, vulnerabilities occurring at the engineering level are a widespread cause of systems' practical insecurity, applying to cryptographic protocols, isolation techniques, and hardware-based trusted computing solutions. This section looks into some but not all of the common ways computers fail to meet the promised security goals.

Design misuses. In 2010, George Hotz managed to break the copy protection feature of the Playstation 3 gaming console due to improper usage of the ECDSA [150]. ECDSA is considered secure by design, but Playstation used ECDSA without proper initialization of the nonce; hence, all generated signatures were using the same value. Recovering the private key from this incorrect usage of ECDSA is as easy as computing two signatures and subtracting them from each other, which entirely breaks the security guarantee of applications and protocols using it.

Developers may mistakenly or willingly omit specific details during the concrete execution of these protocols. Generally, taking the mathematical formulation of cryptographic protocols and transforming it into software or hardware requires domain knowledge and expertise across domains. As a result, the lack thereof such expertise results in misusing a secure design and void security guarantees, which is a widespread issue for cryptographic protocols. In particular, in the past ten years, we have seen several lousy usages of PKC schemes, which shows how this mathematical technique's novelty introduces such uncertainties for developers. For example, for elliptic-curve DH (ECDH), an alternative to DH, an implementation must verify that the public key exchange is a valid point on the expected curve for most elliptic curves. While failure to do this validation would not stall the protocol or damage its quality of service, researchers have shown that omitting this validation step in the code would lead to various attacks [346].

Finding vulnerabilities due to misusing the design may seem harder for proprietary microarchitectures relevant to our work. However, as we will discuss, we have seen that some hardware extensions, such as *Transactional Synchronization Extensions (TSX)* have allowed us to misuse this feature as an amplifier for several microarchitectural attacks [91, 188, 224, 300]. Note that legitimate use cases of TSX and its adoption have been minimal.

Invalid assumptions. Another significant source of vulnerabilities is due to invalid assumptions when transforming abstraction to concrete design and implementation. Therefore, it is vital to understand the execution environments and the attacker's capability before engineering a secure system. This understanding, at a high level, referred to as threat modeling, is vital at every step of making a secure system from the abstract design to low-level implementation. However, we have repeatedly seen that even with proper threat modeling at the abstraction level, the concrete design and implementation suffer from invalid assumptions. These invalid assumptions may have been simply due to oversights, but it worth noting the influence of optimization and cost reductions as an entangled reason.

To provide a few concrete examples, we start with the notion of avoiding security through obscurity. This notion suggests not relying on source code obfuscation or hiding design documents or engineering artifacts as security countermeasures. For example, developers may skip the security review for software, assuming that an attacker can not find vulnerabilities if they do not see the source code. While skipping the security review may prioritize engineering resources, numerous software vulnerabilities on closed-source products prove that this assumption is plain wrong and just an ineffective deterrence for vulnerability hunters and adversaries [294].

Experts have also proposed several cost-effective mitigations for such software vulnerabilities. Some of these cost-effective methods similarly make compromises and weak assumptions rather than resolving software errors' root cause. One example is relying on software diversification methods [108]. We can even argue that runtime diversification techniques are a form of obscurity. Address Space Layout Randomization (ASLR) assumes an adversary can not find the layout of memory pages at runtime; therefore, the exploitation of software defects such as buffer overflows will be hard. However, this difficulty is barely understood; attackers can bypass ASLR by leaking this memory layout, invaliding these assumptions [103, 307].

Additional to making invalid assumptions about the attacker's capability, *i.e.*, weak threat modeling, the complexity of building a system results in invalid assumptions about the execution environment. For instance, the security community suggests developers not to invent their cryptographic software. While this suggestion encourages a better security practice for many use cases, it also pursues the developers into blindly using existing and open source cryptographic software in the wrong context. For example, OpenSSL is a popular cryptographic library [264]. However, OpenSSL developers have not designed it to execute securely for all execution environments and when physical adversaries are concerned.

In the past, we have seen that developers have introduced vulnerabilities by using existing software or components of a system designed for a different threat model in a different environment and threat model. Such a case appeared when OpenWRT routers used Linux's `/dev/urandom` as a source of entropy for cryptographic operations [137]. The Linux random number generator tends not to have enough entropy when executed on this embedded system. Researchers have demonstrated that cryptographic protocols inside this router device are vulnerable as manufacturers have made invalid assumptions about the execution environment for this random number generator, ignoring this use case's particularity.

We will see that invalid assumption, originating from a lack of knowledge and understanding in this space, play roles in some of the microarchitectural vulnerabilities discussed in this dissertation. In particular, rolling out a TEE like SGX based on an ISA not designed initially with a system-level adversarial mindset exemplifies this notion within the hardware computing industry.

Programming vulnerabilities. Engineers tend to optimize their software or hardware logic to perform the best on legitimate inputs. Unfortunately, adversaries do not play with the same rule when it comes to user inputs, and improper handling of unexpected user inputs have caused the creation of several classes of software vulnerabilities, including cross-site scripting [362], code injections [141], and buffer overflows [79]. Handling unexpected inputs is extremely difficult for general-purpose computation such as protocol and file format handlers or scripting environments since the input space is diverse.

Furthermore, the extensive popularity of unsafe programming languages like C and C++ for efficient software has amplified human errors in the form of memory corruption issues like buffer overflow. These programming languages offer high performance for the core application and system software, but they also provide too much flexibility and freedom to developers for introducing input-handling errors. Across the board, these vulnerabilities have affected several of the threat models discussed earlier, giving power to network [79], local [295], and system adversaries [350]. For instance, the *HeartBleed* [94] vulnerability, due to improper bounds checking, allows attackers to steal the private session key from the TLS handshakes remotely. Researchers have proposed several solutions to automate the discovery of these vulnerabilities based on techniques such as input fuzzing [121] and static analysis [73]. Although finding and patching these vulnerabilities and mitigating them with system and compiler-level protections are promising [1], the ultimate pathways to deal with these well-understood security issues are switching to safe programming languages [204] or providing verified and isolated compartments [254].

It is crucial to understand that these solutions for eliminating programming flaws at the architectural level rely on the solidity of the processor's architecture and microarchitectural security. As less understood problems, microarchitectural vulnerabilities can violate security guarantees provided by isolation techniques and programming environments. Efficient and reliable mitigation requires considering microarchitectural flaws as part of the defense in depth. Understanding microarchitectural security is complimentary to building a more robust defense for architectural vulnerabilities.

Side-Channel cryptanalysis and attacks. In 1996, Paul Kocher published a paper about using the timing behavior of operations to attack cryptographic schemes [209]. Although this paper has initiated the past couple of decades of research into side-channel attacks, some people and agencies knew about similar cryptanalysis techniques for a long time. It is not clear when researchers coined the term side channel. Nevertheless, the historical memo from David G. Boak, an educator at the National Security Agency (NSA), implies that defense agencies used similar ideas for eavesdropping and breaking cryptographic machines as early as the *World War II*:

> Across the street, perhaps a hundred feet away, was a hospital controlled by the Japanese government. He sauntered past a kind of carport jutting out from one side of the building and, up under the eaves, noticed a peculiar thing-a carefully concealed dipole antenna, horizontally polarized, with wires leading through the solid cinderblock wall to which the carport abutted. He moseyed back to his headquarters, then quickly notified the counter-intelligence people and fired off a report of this *find* to Army Security Agency, who, in turn, notified NSA. [41]

Most side-channel attacks, as we know them today, exploit the physical behavior of computing systems. During the execution of a computer algorithm, physical signals such as power consumption [228], electromagnetic emanation [12], or merely timing behavior [54] may expose partial information about processed secrets by the algorithm. Side-channel cryptanalysis refers to this usage of partial information recovered through the side channel and combining it with mathematical and algorithmic techniques to break cryptographic schemes. Though, side-channels attacks go beyond cryptanalysis; they are practical for recovering critical information from other components and applications [136, 183, 273].

Intuitively, side channels may not always be due to physical behavior. As for modern digital systems, we expect the communication channels to be through well-defined input and output; hence physical behavior such as power consumption is not part of the execution model and is considered a side channel. *Van Eck phreaking* [355], a technique based on electromagnetic emissions, demonstrates secret recovery in the context of analog TVs a long time ago. Although the idea of using electromagnetic emissions as a side channel in the recent attacks on smart cards is somewhat similar [195], it is not trivial to define what is considered usual channels and side channels for an analog system. With this analysis in mind, we also see modern examples of attacks that may be attributed as side channels or not, depending on very subtle differences. For example, the *Lucky Thirteen* exploits the timing behavior due to CBC-mode AES's decryption failures when a ciphertext includes invalid padding [106]. Although this particular attack relies on a timing side channel, some protocols and applications may simply report decryption failures to the users. A malicious user can use the failure report as an oracle to perform similar cryptanalysis attacks as the *Lucky Thirteen*. Consequently, in this case, depending on how we define the expected communication channels for this cryptosystem, we may call it a side channel. We refer to information leakage through the microarchitectural element's observable timing behavior as side channels in our work.

Side-channel attacks contradict other examples of vulnerabilities mentioned earlier. In those examples, defenders assumed that the computing hardware behaves as expected, and attackers only use defined and expected digital communication channels. This perception is not valid anymore when it comes to side-channel attacks. However, when we try to defend computing systems against these side channels, similar principle problems exist. We may see side-channels leakage due to invalid assumptions, misusing designs, or improper programming and implementation of side-channel countermeasures. As a result, side-channel attacks have created an extra burden by further complicating threat models and engineering side effects. In this context, developers who were previously required to take care of proper threat modeling and engineering at the architecture level now need to know the environmental effects and how the hardware and software layers underneath behave.

1.2 Microarchitectural Security

Microarchitectural attacks. The first side-channel attacks that exploit CPU's cache's timing behavior were proposed more than a decade ago [32, 267, 269]. However, microarchitectural security and software-based side-channel attacks have more recently become a trending avenue of research. Microarchitectural attacks have evolved over the past decade from attacks on weak cryptographic implementations [32] to devastating attacks breaking through layers of defenses provided by the hardware and the Operating System (OS) [348]. These attacks can steal secrets such as cryptographic keys [31, 270] or keystrokes [222]. More advanced attacks can entirely subvert the OS memory isolation to read the memory content from more privileged security domains [224], and to bypass defense mechanisms such as Kernel Address Space Layout Randomization (KASLR) [103, 126]. Rowhammer attacks can further break the data and code integrity by tampering with memory contents [202, 304]. While most of these attacks require local access and native code execution, various efforts have been successful in conducting them remotely [331] or from within a remotely accessible execution environment like JavaScript [265].

One crucial difference between microarchitectural attacks and previous hardware vulnerabilities [194], and traditional physical side-channel and fault attacks [37, 229] is the ability for adversaries to exploit them from the software. As a result, vulnerabilities stem from software-based side channels, and microarchitectural attacks affect a wide range of threat models and products. For instance, network adversaries can exploit these vulnerabilities through drive-by web-based attacks to steal cryptographic keys [117]. Local adversaries can exploit these vulnerabilities to break process-level and VM-level isolation [184, 224, 300]. Worst, in the trusted computing model, attackers can tweak these attacks for more efficient and effective data exfiltration, violating privacy and security promises of TEEs [249, 348, 349].

In response to recent discoveries in this domain, researchers have also proactively proposed countermeasures and mitigation. These countermeasures cover a diversity of techniques and research paradigms. Cryptographers have proposed design and implementation choices that avoid secret-dependent memory accesses [34] or hide these access patterns by following constant-time techniques [197]. Researchers have also proposed several automated compiler-based mitigations for some of these attacks [277, 363]. Several proposals work at the system-level in which the operating system is responsible for mitigating these attacks through specific memory management policies and techniques [48, 379]. Some other proposals suggest runtime attack detection by using CPU's performance counters [253]. Ultimately, the microarchitecture community has proposed fundamental changes to eradicate some of these attacks applicable to future processors [87, 198].

Challenges. These attacks and countermeasures introduce sophisticated engineering challenges that the industry has barely seen to adopt. In fact, microarchitectural security is in its fancy. As researchers and engineers, we do not have a proper understanding of how and if we should mitigate various attack vectors in this domain. We have identified three main problems that contribute to this lack of understanding:

1. **Earliness:** Previous defensive efforts focus on a single attack vector e.g., cache attack. However, we do not have full coverage of all the possible attack techniques. Even on a ubiquitous microarchitecture like the Intel Core generations of CPUs, other researchers before us and we have found new attack vectors every year in the past decade. For microarchitectures that are evolving rapidly, understanding potential attack vectors will be even more difficult. This difficulty is beyond existing classes of vulnerabilities like buffer overflows and cross-site scripting on the web that are generally well understood.

2. **Fuzzy impact:** Unlike software vulnerabilities, cryptographic flaws, or physical attacks, understanding the impact of microarchitectural vulnerabilities depends on several external factors. A subtle microarchitectural vulnerability may have no impact or severe impact, depending on the software environment and execution context. For example, Flush+Reload [384] is a cache attack technique that requires access to shared read-only memory pages. We have seen researchers using this technique to attack cryptographic implementations. Still, we do not know how common it is for products at both enterprise and standard level to use shared memory pages across security domains. While this complexity may suggest that Flush+Reload has no practical impact, we saw later that Flush+Reload amplifies other microarchitectural vulnerabilities like Spectre [207] and Meltdown [224] and software vulnerabilities in the Linux kernel [298]. This difficulty in understanding the real impact is due to modern software's complexity and the lack of research into these vulnerabilities' practical implications on software systems.

3. **Limited expertise and tooling:** Both of the above problems suggest that we need to analyze microarchitectures for finding and understanding attack vectors and studying and demonstrating the impact of these vulnerabilities on the software ecosystem. Since both of these efforts are heavily labor-intensive, expertise and tooling would be tremendously helpful. However, as a new research topic, microarchitectural security is only mastered by a few experts. There is also a lack of tooling to aid people with less expertise to learn and practice microarchitectural security.

These problems collectively contribute to microarchitectural security being a less understood field. As a result, proposed mitigation often addresses these attacks individually or applies to a particular instance of these attacks. More importantly, since we do not understand some attacks' impacts, there is no clear direction for the industry on spending both engineering and performance costs into defending against these sophisticated attacks.

The performance cost is indeed oversimplified for some of the academic defense proposals. Fixing microarchitectural vulnerabilities is unlike mitigating software vulnerabilities with a limited scope, and it results in changing several performance metrics. For example, Spectre's real impact on different products is still an open research problem. Despite this unresolved understanding of the real impact, proposed hardware mitigations would drastically kill the entire software stack's performance, not only a single software or protocol [65]. In the big picture, the real question is if it is worth paying such tremendous performance overhead to mitigate all of them. In this context, ad-hoc approaches to solve these problems lack enough vision and practicality. The real question is *what are we even trying to defend against when we do not understand the hardware's underlying complexity, attack vectors, and their impact on weakening security boundaries?*

Proposal. In this work, we aim to tackle some of the above problems in three directions: **(I)** We study memory subsystems' behavior on the commodity Intel Core CPUs from a security perspective through black-box reverse engineering and microbenchmarks. This step enables us to develop insights and tools for microarchitectural-security analysis and find new attack vectors. These attack vectors will help us and the community to have a better understanding of the vulnerability landscape. **(II)** We design and develop tools to automate discovery and analysis of some of the vulnerabilities. We hope that with automated analysis, we can better understand the root cause and impact of such vulnerabilities. **(III)** For every new attack vector that we discover, we demonstrate attacks on several cryptographic implementations known to be resistant against previous attacks. To achieve end-to-end demonstration, we apply state-of-the-art mathematical cryptanalysis techniques and devise new algorithmic methods.

We follow this three-pronged paradigm in the context of several threat models, including network adversaries, local adversaries, and system adversaries. In summary, we propose these techniques under the umbrella of *microarchitectural cryptanalysis*. We hope that our work will become a standard procedure for the security testing of computing systems processing highly-sensitive assets.

1.3 Contributions

We started looking into microarchitectural side channels within the context of TEEs as part of my master thesis's project [243]. This thesis resulted in *CacheZoom* [245], a high-resolution cache attack against SGX, in collaboration with Gorka Irazoqui and Thomas Eisenbarth. CacheZoom highlights that attack vectors like the cache side channel have a higher impact on TEEs, and attackers can perform cryptanalysis of software-based encryption schemes like AES more efficiently. Motivated by this earlier work, during my Ph.D. work, we continued working in this domain by finding new attack vectors affecting various threat models, improving the state-of-the-art tools and techniques, and demonstrating end-to-end cryptanalysis attacks against several cryptographic software. These findings were presented in close engagement with industry partners and other academics through coordinated responsible disclosure, research talks and publications, and joint projects, essentially increasing awareness and improving the affected products' security. We provide an overview of this dissertation's main contributions in Section 1.3.1. We have also contributed to several other publications during my Ph.D. Although we have not included them in this dissertation, for completeness, an overview of these additional contributions is provided in Section 1.3.2.

1.3.1 Main Contributions

MemJam. The effect of potential intra-cache-line microarchitectural behaviors like false dependencies was not considered a practical security threat on recent microarchitectures [385]. Consequently, developers have deployed several ad-hoc countermeasures, dubbed constant-time techniques, to avoid secret-dependent cache access patterns during the execution of operations such as encryption or signature generation. In MemJam [244], in collaboration with Thomas Eisenbarth and Berk Sunar, we precisely analyzed the 4K aliasing false dependency across sibling CPU threads. Our analysis results in a new intra-cache-line side-channel attack bypassing proposed ad-hoc countermeasures for cache attacks. In contrast to cache bank conflicts, MemJam affected all Intel Core generations known at the time of this study. Consequently, we demonstrated several cryptanalysis attacks against implementations of AES and SM4 that were presumed to be secure. We later extended this analysis to 3DES in collaboration with Jan Wichelmann, essentially showing that all software-based encryption schemes inside Intel IPP cryptographic library were vulnerable to MemJam [246].

Spoiler. Based on our understanding of MemJam, in Spoiler [187], in collaboration with Saad Islaam, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar, we further analyzed the CPU's memory subsystem and addressed an undocumented false dependency behavior. This time, we showed that the memory subsystem exposes critical timing behaviors depending on physical address bits. Consequently, we managed to leak more information at runtime about the physical layout of memory pages. We leveraged this information to boost previous microarchitectural attacks like the cache Prime+Probe and rowhammer. Our work shows the importance of precise analysis and reverse engineering of commodity microarchitectures. This original analysis indicates that core-private leakages contribute to more powerful attack vectors on off-core components such as the shared last-level cache and DRAM.

Transynther and Medusa. Based on the revealed knowledge about how the memory subsystem works on superscalar CPUs, we have contributed to several new transient execution attacks based on Meltdown as part of collaborations with other researchers (§1.3.2). However, the analysis and discovery of Meltdown-style vulnerabilities is a stochastic and error-prune problem. We designed a tool, named Transynther [247] based on fuzzing techniques, that automatically synthesize and analyze code snippets for *microarchitectural data sampling (MDS)*. This tool aims to help find new variants of MDS, synthesize code snippets for attack demonstration, and ultimately help with automated testing of future microarchitectures. Using Transynther, we provide new insight into Meltdown attacks' root cause and disclose new exploitation methodologies. In particular, the Medusa attack exploits implicit write-combining memory operations, e.g., `rep mov`. To show the impact of Medusa on real-world software, we combine the data leakage from Medusa with state-of-the-art Coppersmith's technique. As a result, we demonstrate an end-to-end attack on an RSA implementation known to be secure against all previous side-channel and microarchitecture attacks.

We have published the open-source Transynther tool, Medusa technique, and the RSA attack demonstration at the Usenix security conference proceedings, in collaboration with Moritz Lipp, Berk Sunar, and Michael Schwarz. Later, we also tested Transynther on more recent Intel CPUs, claiming to be secure against MDS attacks. Our tool discovered that 10_{th} generation Intel CPUs does not adequately mitigate one of the MDS variants, which confirms our tool's usefulness and how automated testing is crucial for microarchitectural security analysis.

CopyCat. While most of our findings affect multiple threat models, including local adversaries and system adversaries attacking SGX, based on our prior work [244, 245, 300], we have learned that microarchitectural attacks have a more severe impact on the system adversarial model. More importantly, adversaries can develop new, unusual attacks like the state-of-the-art controlled-channel to target the SGX threat model [380]. Controlled-channel attacks exploit the ISA's architectural features to exfiltrate memory-access patterns and runtime control flow in a deterministic fashion, *i.e.*, without measurement noise.

In CopyCat [249], in collaboration with Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar, we devise a new attack based on the SGX-Step [380] framework that improves the spatial resolution of controlled channel attacks from 4 kB granularity to instruction-level granularity. CopyCat is a novel controlled-channel attack that leak runtime control flow from Intel SGX enclaves without noise at an instruction-level granularity. We explore the impact of CopyCat on general-purpose use cases by defeating a state-of-the-art compiler hardening technique against branch shadowing attacks. The technique demonstrated by CopyCat bypasses several previous countermeasures that assume adversaries can only perform controlled-channel attacks with page-level resolution. This attack differs from previous attacks using cache and branch predictor's leakage, which can only probe locally, and they suffer from noise. CopyCat opens a new avenue of single-trace attacks on runtime control flow that can not be mitigated by page-level obfuscation or tweaking and isolation of microarchitectural buffers and components.

In an extensive empirical case study of side-channel vulnerabilities in widely-used cryptographic libraries including WolfSSL, Libgcrypt, OpenSSL, and Intel IPP, we verify the practicality and capability of these attacks, demonstrate several attacks, and report vulnerabilities in some of these libraries. We devise new algorithmic techniques to exploit these vulnerabilities in DSA, ECDSA, and ElGamal and RSA key generation, which result in complete key recovery in the context of Intel SGX.

TPM-Fail. Although physical isolation of trusted elements promises strong guarantees compared to TEEs, in TPM-Fail [248], we show that they are not immune to side-channel and timing attacks. We perform a black-box timing analysis of TPM devices using microarchitectural timing analysis. Our analysis reveals that elliptic curve signature operations on TPMs from various manufacturers are vulnerable to timing leakage, leading to the private signing key's recovery. We show that this leakage is significant enough to be exploited remotely by a network adversary.

TPM-Fail, in collaboration with Thomas Eisenbarth, Berk Sunar, and Nadia Heninger, suggests an analysis tool that can accurately measure TPM operations' execution time on commodity computers. As a result, we discover previously unknown vulnerabilities in TPM implementations of ECDSA and ECSchnorr signature schemes and the pairing-friendly

BN-256 curve used by the ECDAA signature scheme. We apply lattice-based techniques to recover private keys from these side-channel vulnerabilities. We also demonstrate a remote attack that breaks the authentication of a VPN server that uses Intel fTPM to store the private certificate key and sign the authentication message. We demonstrate our attack's efficacy against the strongSwan IPsec-based VPN Solution that uses the TPM device to sign authentication messages. Our study shows that these vulnerabilities exist in devices validated based on FIPS 140-2 Level 2 and Common Criteria (CC) EAL 4+, the highest internationally accepted assurance level in CC, in a protection profile that explicitly includes timing side channels.

1.3.2 Other Contributions

Cryptographic implementations. In *CacheQuote* [82], we apply the cache side channel of CacheZoom to analyze the security of Intel's EPID Protocol, as implemented inside SGX's quoting enclave. We show that we can recover the SGX enclave's long term key partially due to how Intel's implementation of the EPID protocol leaks the length of the randomness used for one of the zero-knowledge proofs of EPID. To exploit this leakage, we extend the *hidden number problems (HNP)* to this zero-knowledge protocol. This exploitation allows a malicious attestation server operator to break the unlinkability guarantees of SGX's remote attestation protocol. In addition to this practical demonstration, we also show experimental evidence that the lattice attack can still succeed even when we observe a small number of erroneous traces. CacheQuote essentially shows that even known attack vectors like CacheZoom are hard to prevent for sophisticated software. The vendor who was aware of this attack vector still deployed the EPID cryptographic library without considering such subtle leakage of secrets.

Since detecting vulnerability of cryptographic implementations to microarchitectural side channels and locating the vulnerable part of the code is challenging, in MicroWalk [368], in collaboration with Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar, we propose a leakage detection technique and develop a framework to locate leakages within software binaries. MicroWalk is extendable to locate other and new types of microarchitectural leakages. It is a technique based on dynamic binary instrumentation and mutual Information analysis to find memory-based and control-flow based microarchitectural leakages in software binaries. We demonstrate the ease-of-use of MicroWalk by showing how it significantly eases binary analysis even when source code is not accessible to the analyst. We apply MicroWalk to cryptographic schemes implemented in Microsoft CNG and Intel IPP, both widely used yet closed source crypto libraries. Our results include previously unknown leakages in these libraries, quantification of the critical leakages, and discussing the security impact of these leakages on the relevant cryptographic schemes. MicroWalk is an automated approach that would help engineers to avoid vulnerabilities in cryptographic implementations such as those demonstrated in MemJam, TPM-Fail, CacheZoom, and CacheQuote.

Transient execution attacks. We started looking into applying the Meltdown technique to other microarchitectural elements based on how our understanding of Intel CPUs handle data dependencies studied in MemJam and Spoiler. In *Fallout* [61], we show that data leakage is still possible even on newer Intel hardware, which skips software-based countermeasures like the kernel page table isolation. At the microarchitectural level, in this work, we focus on the store buffer. This microarchitectural element serializes the stream of stores and hides the latency of storing values to memory. In summary, Fallout contributes a new security flaw due to specific shortcuts in Intel CPUs that allow us to leak the data corresponding to recent memory stores. We demonstrate these behaviors' security implications by recovering the values of recent stores performed by the OS kernel, leaking cryptographic keys, and breaking the KASLR mitigating mechanism.

In *ZombieLoad* [300], we further analyze other root causes for the exploitation of Meltdown and focus on another microarchitectural element, called the line fill buffer (LFB). In contrast to Fallout, ZombieLoad leaks data from sibling CPU threads as Intel CPUs share the LFB among multiple threads running on the same CPU core. We combine random data sampling in the time domain with traditional side-channel primitives to construct a targeted information flow similar to regular Meltdown attacks. We demonstrate ZombieLoad in several real-world scenarios: cross-process, cross-VM, user-to-kernel, and SGX. We show that ZombieLoad breaks the security guarantees of Intel SGX, even on Foreshadow-resistant hardware.

ZombieLoad and Fallout work on Meltdown-resistant hardware. These vulnerabilities under the umbrella of microarchitectural data sampling were found and analyzed in collaboration with researchers from Graz University of Technology, Katholieke Universiteit Leuven, University of Michigan, and the University of Adelaide. Other industry researchers and academics from Vrije Universiteit Amsterdam and CISPA Helmholtz Center for Information Security discovered similar behaviors.

In another contribution, *Load Value Injection (LVI)* [349] shows that Meltdown-type data leakage can be inverted into a Spectre-like Load Value Injection (LVI) primitive. LVI transiently hijacks data flow, and thus control flow and presents an extensible taxonomy of LVI-based attacks. We show the insufficiency of silicon changes in the latest generation of acclaimed Meltdown-resistant Intel CPUs. As a result, we develop practical proof-of-concept exploits against Intel SGX enclaves, and we discuss implications for traditional kernel and process isolation in the presence of suitable LVI gadgets and faulting or assisted loads. Our evaluation of compiler mitigations for LVI suggests that native and wholesome mitigation incurs a runtime overhead of factor 2 to 19. LVI creates a new avenue of research into compiler-based optimization and defense for legacy SGX hardware that will not benefit future hardware designs.

Future explorations. Microarchitectures are becoming more heterogeneous, integrating accelerators such as GPUs, FPGAs, and AI accelerators. Consequently, we expect to see more microarchitectural attacks. As a pioneer in this direction, in *JackHammer* [367] in collaboration with Zane Weisseman, Thore Tiemann, Evan Custodio, Thomas Eisenbarth, and Berk Sunar, we demonstrate novel attacks between the memory interface of Intel Arria 10 GX platforms and their host CPUs. In summary, we thoroughly reverse-engineer and analyze the cache behavior and investigate the viability of cache attacks on realistic FPGA-CPU hybrid systems. Based on our study of the cache subsystem, we build JackHammer, a Rowhammer from the FPGA, that bypasses caching to hammer the main memory. We compare JackHammer with the CPU Rowhammer and show that JackHammer is twice as fast as a CPU attack, causing faults that the CPU Rowhammer cannot replicate. JackHammer remains stealthy to CPU monitors since it bypasses the CPU microarchitecture. Using both JackHammer and conventional CPU Rowhammer, we demonstrate a fault attack on recent RSA implementation versions in the WolfSSL library and recover private keys. JackHammer shows how combining independent microarchitectures into the same memory subsystem introduces new security and privacy challenges.

Another avenue of future explorations is the use of deep learning and artificial intelligence in this domain. In collaboration with Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar, we propose *FortuneTeller* [135] which is the first generic detection model for microarchitectural attacks. FortuneTeller learns the system's behavior by observing microarchitectural events and classifying outliers not conforming to the trained model. FortuneTeller detects unseen microarchitectural attacks since it only requires training over benign execution patterns. In summary, this is a generic detection technique that can be applied to detect attacks on other microarchitectures and execution environments can automatically detect various attacks, disregarding the victim application, including cryptographic implementations, browser passwords, private data in the kernel environment, bit flips, and so on. FortuneTeller combines hardware performance counters with advanced Recurrent neural network algorithms by training a more refined and generic model. We show how this combination performs better by comparing it to the state-of-the-art microarchitectural detection techniques.

1.4 Outline of the Work

Chapter 2 focuses on microarchitectural side-channel attack vectors. We extensively discuss our contributions to microarchitectural side channels stemming from memory dependency analysis. In Chapter 3, we particularity focus on transient execution attacks that leak data from the CPU by discussing prior work, TRANSYNTHER, and the MEDUSA. Chapter 4 provides a taxonomy of attacks in the system-adversarial model against SGX and ultimately cover the COPYCAT contribution. In Chapter 5, we will focus on another hardware-based trusted computing technique and discuss our findings in TPM-FAIL. Chapter 6 puts together what we have learned about side-channel and data leakage. We combine these attack vectors with cryptanalysis techniques under the framework of microarchitectural cryptanalysis. We report several end-to-end microarchitectural cryptanalysis attacks. Ultimately, in Chapter 7, we put all of our findings and previous work on attack and defense into the big picture and provide conclusions and recommendations in this domain.

Chapter 2

Uncovering Microarchitectural Side Channels

Resource contention within the microarchitecture may occur when independent programs share the same microarchitectural components. Modern central processing units (CPUs) are optimized to avoid such contentions as much as possible during the execution of regular workloads. However, an adversary can trigger such resource contentions intentionally with a specially crafted program.

This chapter first provides some background information about the CPU microarchitecture (§2.1). Next, we provide an overview of the proposed works in the literature that highlights several security issues due to such intentional resource contentions (§2.2). In Section 2.3, we perform an in-depth analysis of false dependencies between memory operations issued by separate CPU threads. As a result, we propose an attack named MemJam that exploits false dependencies due to partial address aliasing to construct a novel side channel with high spatial resolution. MemJam highlights the importance of scrutinizing the microarchitecture's security analysis while bypassing several countermeasures against cache attacks. In Section 2.4, we advance this analysis further by discovering and exploiting a new microarchitectural behavior related to addressing logics. This time, our proposed attack, named Spoiler, shows how physical-address leakage within the CPU core has security implications for shared caches and DRAM. Finally, Section 2.5 summarizes our findings.

2.1 CPU Microarchitecture

Modern CPUs connect multiple cores to a shared *last-level cache (LLC)* and the main memory, *i.e.*, DRAM via a coherent memory subsystem. Each core can execute at least

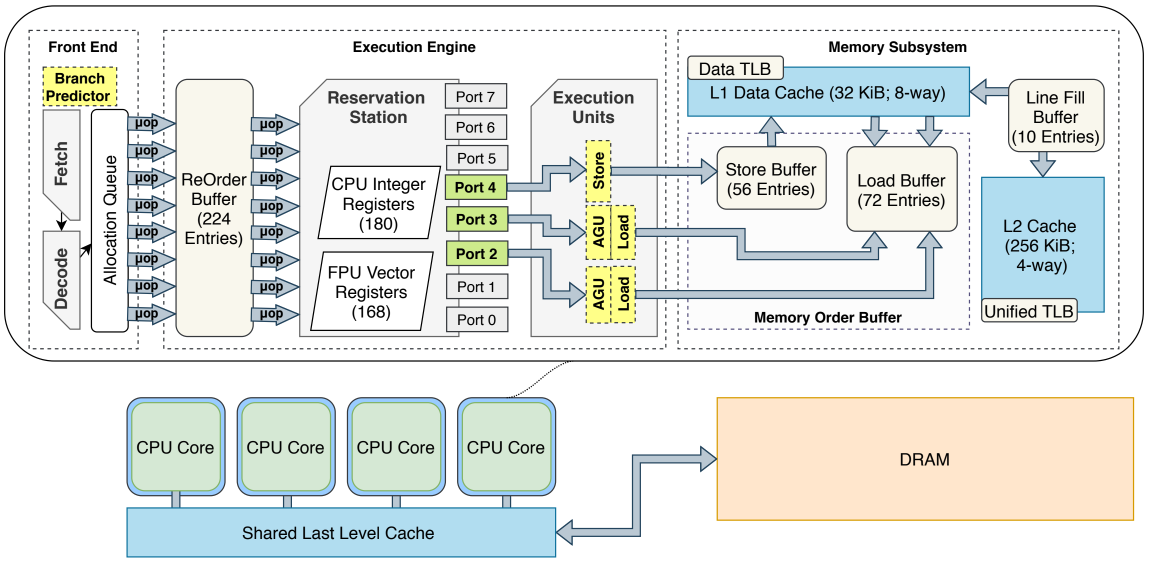


Figure 2.1: Schematic of a superscalar CPU. Multiple cores are connected to the DRAM and a shared LLC via an interconnect. The front end of the core fetches and decodes instructions into μOPs. The execution engine assigns ROB entries for each μOPs, allocating resources to execute μOPs. Although the execution of μOPs is out-of-order, the ROB is responsible for in-order completion and retiring instructions. Memory operations will load and store data from and to the next levels of cache and, ultimately, the DRAM through the memory subsystem and several internal data and address buffers. This subsystem includes store buffer, load buffer, line fill buffer, and translation lookaside buffer (TLB).

a single software thread or multiple threads if it supports simultaneous multithreading (Section 2.1.3). Figure 2.1 illustrates the internal of a CPU core and its placement concerning the LLC and DRAM. Each core executes programs through a multi-stage pipeline, e.g., 14-19 stages on Intel Skylake, and synchronizes operations by adopting out-of-order and speculative execution techniques. These pipeline stages are virtually scattered within the *front end*, the *execution engine* and the *memory subsystem*. The front end fetches and decodes program instructions, and the execution engine allocates resources and executes these instructions in collaboration with the memory subsystem. Next, we discuss several components of the CPU core, the memory subsystem, and related optimization techniques.

2.1.1 Out-of-order and Speculative Execution

Fetch and decode. The front end fetches program instructions from the instruction cache and places them into the allocation queue. In a complex instruction set computer

(CISC) such as Intel, each instruction is first decoded to smaller micro-operations (μOPs) using a complex decoder or a microcode sequencer [63]. Decoded μOPs may also be wang2017cached in a μOP cache to speed up the decoding of frequently executed instructions. For simplicity, Figure 2.1 omits some of these details.

The core fetches a long sequence of instructions ahead of time, but it also brings instructions that are not sequentially available in the program with the branch predictor's help. The branch predictor maintains the history of the previous branch instructions' execution and predicts based on their record. Generally, the branch predictor, based on some proprietary design, chooses a target address for an indirect branch, e.g., `jmp <register>`, and it also predicts the binary decision of conditional branches, e.g., `jCC [target]`. As a result, branch predictor helps the core not stall on latent control flows that are not entirely executed (retired) by facilitating speculative fetching and execution of instructions.

Execution engine. In addition to fetching instructions ahead of time, to benefit from a deep out-of-order pipeline, the core optimizes its resources by synchronously allocating resources and executing multiple instructions at a time. The execution engine retrieves the μOPs from the allocation queue, then allocates resources while the instructions are waiting in the reorder buffer (ROB) to be completed. In parallel, the scheduler sends the μOPs to various execution units depending on the availability of resources. The CPU core executes these μOPs out-of-order, and some may succeed before preceding instructions.

While the correct execution of instructions may depend on those prior instructions, the CPU transiently executes such instructions if they meet all dependencies. The ROB will retire completed μOPs without failures according to the correct ordering instructions inside a program. Generally, the core only verifies the architectural consistency right before committing the architectural registers and memory results. If the ROB detects an error for an operation, it will discard the wrong outcome, *i.e.*, the pipeline is flushed, and corresponding μOPs will be rescheduled. Instructions that perform memory operations access the memory through the memory subsystem.

Speculative execution. As mentioned, the pipeline can fetch instructions ahead of time and also execute them out-of-order. During out-of-order execution, the core may execute an instruction based on speculation about the outcome of preceding instructions. For instance, during a `load` that executes before preceding stores, the core may predict a dependency between the `load` and preceding `stores` and performs the load operation based on this prediction. Later, if this prediction outcome turns out to be wrong, the pipeline will flush and re-execute the `load` and its dependent instructions [146, 187]. Similarly, thanks to the branch predictor, branches are executed speculatively based on predicting the target and direction of the branch [219]. Speculative execution can generally refer to any operation that relies on a form of a prediction. While speculative execution improves resource usage within the core and speeds up applications by avoiding unnecessary stalls, it has been the cause of several microarchitectural vulnerabilities [187, 207, 211].

2.1.2 Memory Subsystem

DRAM memory is slow compared to the internal CPU components. Programs running on the CPU tend to access the same data and instruction repeatedly. However, accessing memory to get the same information takes too much time, which causes a bottleneck in the system. Modern microarchitectures have multiple caches and buffers to fill the speed gap between execution units and the DRAM and speed up accessing information. Before discussing the components within the memory subsystem, it is essential to understand *virtual addresses* and *address translation*.

Address translation. On a superscalar CPU, programs only use virtual addresses to execute code and access data, while the memory operations are mapped to the DRAM using physical addresses. The virtual memory manager of the OS shares the DRAM across all running tasks by assigning isolated virtual address spaces to each task. Each task can use its entire virtual address space without the meddling of memory accesses from others. The system assigns memory regions in *page* granularity, which is generally 4 kB each. Each virtual page will be stored as a physical page in DRAM through a virtual-to-physical page mapping. CPUs generally support larger page sizes for specific use cases when a program or a device driver requires access to contiguous physical memory space.

The CPU core translates the virtual addresses of code and data pages to physical addresses with the operating system's help. Figure 2.2 demonstrates the translation process, including usage of a translation lookaside buffer (TLB) and page miss handler (PMH). For the translation process, the OS maintains a hierarchy of page tables. The lower 12 bits of a virtual address, corresponding to the offset within a 4 kB page, are directly mapped to physical address space. The upper bits are subjected to the translation using a page-table hierarchy that is maintained by the OS. Ultimately, the page-table hierarchy translates the upper bits to a page table entry (PTE). The PTE of each page consists of some metadata and the physical page number. Since translation using the page-table hierarchy is time-consuming, a translation lookaside buffer (TLB) stores the translations for recently-accessed pages. If a virtual address translation is not present in the TLB, then the PMH triggers the OS to perform its page walk procedure to obtain the physical address. As we will discuss, the virtual to physical address translations add **complexity** to other memory subsystems with potential **security concerns**.

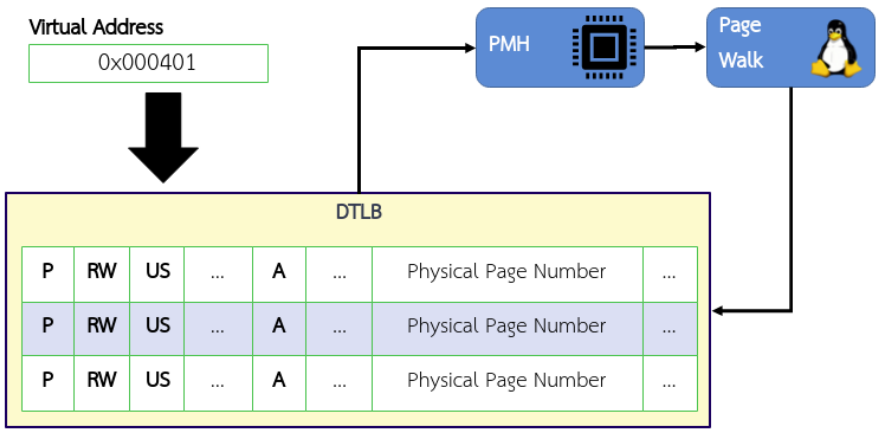


Figure 2.2: To translate a virtual address, the CPU first consults the TLB. If the translation (PTE) is not present in the TLB, the CPU interrupts the OS to perform a page table walk.

Cache hierarchy. Intel CPUs have two levels of core-private cache (L1, L2) and the shared last level cache (LLC). The closer the cache memory is to the CPU, the faster and smaller it is than the next level cache. The LLC, shared across CPU cores, is connected through an interconnect bus to the DRAM, peripherals, and other subsystems. Both the LLC and L2 cache are unified caches, *i.e.*, they hold both data and instruction cache lines. The L1 cache is composed of separate units for storing data (L1D) and instructions (L1I); the L1D is for data cache lines, and the L1I is for instruction cache lines.

The cache memory consists of multiple slices and sets. In a set-associative cache like on Intel CPUs, each set stores a certain number of cache lines (ways). The size of a cache line, 64 bytes, is the block size for all memory transactions across caches and the DRAM. In general, Intel CPUs have 64 sets for the L1 cache, and the number of ways is 8. Therefore, the cache subsystem uses the six least significant bits of the virtual address (same as physical) to determine the offset within a line—the remaining bits to pick the set to store the cache line.

The number of physical address bits used for mapping is higher for the LLC since it has many sets, e.g., 8192 sets. Hence, the untranslated part of the virtual address bits, which is the page offset, cannot index the LLC sets. Instead, the indexing uses higher physical address bits. Further, the cache subsystem divides LLC sets into multiple slices, one slice for each CPU core. The mapping of the physical addresses to the slices uses an undocumented function, which previously has been reverse-engineered for some microarchitectures [183].

Cache operations. When the CPU tries to access a cache line, a cache hit or miss occurs respective of its existence in the relevant cache set. If a cache miss occurs, the CPU brings the target memory line to all cache levels and the determined sets. Reloads from the same address would be much faster when the memory line exists in the cache. The cache serves accesses to the same memory address unless other memory accesses evict that cache line. We can also use the `clflush` instruction, which follows the same memory access check as other memory operations to evict our cache lines from the entire cache hierarchy.

In a multicore system, the CPU also keeps cache consistent among all levels. On most architectures, cache lines by default follow a write-back policy, *i.e.*, if an instruction overwrites the data in the L1 cache, it will propagate to all other cache levels. The LLC is inclusive of L2 and L1 caches, which means that if an operation evicts a cache line in LLC, the CPU expels the corresponding L1 and L2 cache lines [163]. These policies help to avoid stale wang2017cached data where one CPU reads invalid data mutated by another CPU. Set-associative caches also require to support a replacement policy that determines the order to evict a cache line when the set becomes full. For example, a standard replacement policy is the least recently used (LRU) policy. The detail for the replacement policy is not relevant to our work.

Line fill buffer. The CPU uses a fill buffer to service memory accesses missing the L1 cache. When an L1 cache miss occurs, the CPU will allocate an entry inside the fill buffer to collect the data bytes from the next cache or the DRAM. Note that it may forward data from the fill buffer to memory loads before filling the entire cache line. The fill buffer can also temporarily service the data for memory accesses of uncachable (UC) type. UC memory commonly used by device drivers for memory-mapped IO (MMIO) is directly serviced from the main memory and will bypass the cache. We will discuss different memory types and their setting shortly. In general, fill buffers assure that the entire memory bytes corresponding to a memory instruction are available before forwarding it to load operations. On some microarchitectures, the memory subsystem deallocates 'fill buffer' by allowing other memory operations to use a freed entry. Consequently, the old/stale data may stay in the buffer until another memory operation overwrites it. Microarchitectural data sampling showed that Intel CPUs might forward stale data to malicious load operations [300].

Memory types. CPUs support multiple per-page memory types with different policies for caching and ordering guarantees. The software can set these memory types for every physical page at page-level granularity. The supported memory types on x86 are write-back (WB), write-through (WT), write-protect (WP), write-combining (WC), and uncachable

(UC). Most pages are write-back, which allows them to be wang2017cached and written back to memory later. Both UC and WT write data directly to memory. In contrast, WT also caches the data but writes it back to the memory synchronously. Uncachable memory is never wang2017cached and directly written back to the memory. Uncachable memory is required for memory-mapped devices to ensure that the data reaches an external subsystem as soon as the CPU writes it to the DRAM and with the right byte ordering. Writing-combining memory tries to reduce bus requests by combining multiple stores to the same cache line.

A memory store has to update core-private caches, the LLC, and possibly the main memory. Thus, for performance, it is beneficial to combine multiple stores into a single request. This technique, known as write combining (WC), reduces the number of bus requests and cross-core snoops that update the core-private copy of the cache. With WC, the CPU temporally buffers the store operations' data to the same cache line until all the memory bytes that modify that cache line are available. Some CPUs, like in AMD CPUs, implement WC with a dedicated buffer [10]. In Intel CPUs, WC allocates an entry from the fill buffer [177]. Programs often use WC for memory operations in places where memory ordering guarantees are weak, e.g., for frame buffers of graphic cards, which are usually treated as write-only by programmers [171].

There are two different mechanisms to set the memory type for a page. The legacy method is to use one of the memory type range registers (MTRRs). These registers allow setting the memory type for a physical address range. The modern alternative is to use page-attribute tables (PATs). With PATs, the operating system can define and concurrently use a list of 8 memory types. Typically, this list contains all available memory types plus the memory type *UC-*. UC- is the same as uncachable, with the only difference that it is flexible to changes to a different kind using an MTRR range. Every page-table entry contains three bits which select the corresponding entry from the PAT and apply a memory type to that page [177].

Memory order buffer. Intel CPUs manage memory operations inside the memory order buffer (MOB). While MOB is mostly a terminology from Intel, other superscalar CPUs generally follow a similar technique, as explained below. MOB works closely with the data cache to assure that memory operations are executed efficiently by following the memory ordering rule [172]. This rule implies that the CPU must execute store operations in-order and load operations out-of-order. The memory subsystem should enforce these rules to improve memory access efficiency while guaranteeing their correct commitment to cache and DRAM.

Figure 2.3 shows the MOB schematic according to Intel [2, 3]. The MOB includes circular buffers, *store buffer* and *load buffer* (LB). Store buffer consists of *store address* *buffer (SAB)* and *store data buffer (SDB)*. We use *store buffer* to mention the logically combined SAB and SDB units for simplicity. A store operation will be decoded into two μOPs to store the address and data, respectively, to the store buffer. The store buffer enables the CPU to continue executing other instructions before the commitment of preceding store operations. As a result, the pipeline does not have to stall for these stores to complete. This logic further enables the MOB to support out-of-order execution of the load operations.

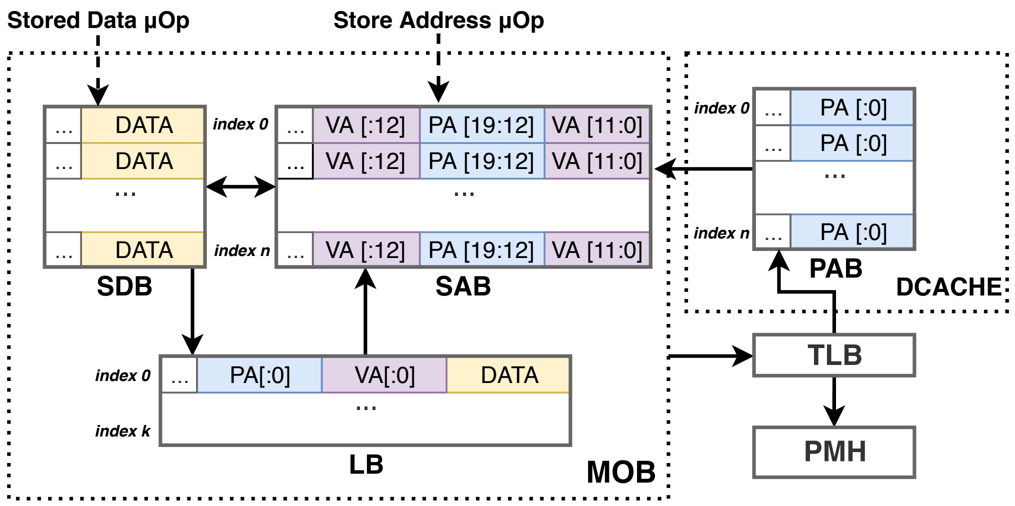


Figure 2.3: The memory order buffer includes circular buffers SDB, SAB, and LB. SDB, SAB, and PAB of the data cache have the same number of entries. SAB may initially hold the virtual address and the partial physical address. MOB requests the TLB to translate the virtual address and update the PAB with the translated physical address.

The CPU core incorporates several optimization techniques within the MOB such as *speculative loads* [92], *store forwarding* and *memory disambiguation* to improve the memory bottleneck. The CPU can generally execute memory load operations faster than the store operations since stores may need to update multiple caches and the DRAM. A memory load may bypass preceding stores to avoid pipeline stalls due to the potential false dependency of `load` on `stores`. On the other hand, it may forward data from the store buffer to the `load` when necessary. Either way, the load operation has to execute speculatively.

Store forwarding. Store forwarding is an optimization technique that sends the data from the store buffer to a memory load operation if the `load` address matches any of the store buffer entries. This optimization is a speculative process since the MOB cannot determine the actual dependency of the `load` on `stores` based on the store buffer. Intel's implementation of the store buffer is undocumented, but a potential design suggests that it will only hold the virtual address, and it may include part of the physical address [2, 3, 212]. As a result, the CPU may falsely forward the data, although the physical addresses do not match. The ROB delay the ultimate resolution until the `load` commitment since the MOB needs to ask the TLB for the complete physical address information, which is a time-consuming operation. Additionally, the L1D cache may hold the translated store addresses in a physical address buffer (PAB) with an equal number of entries as the store buffer. This enhances performance since the `load` does not have to wait for preceding `stores` to complete. However, we can not confirm if PAB exists on the Intel products. In general, the dependency prediction may rely on partial address information, leading to false dependencies and stall hazards.

Memory false dependencies. Dependency prediction and resolution logic circuits are in place to determine if a `load` is dependent on any of the preceding store buffer entries. During store forwarding or load bypass, the CPU may fail to predict or resolve the correct dependencies between the `load` and `stores` [187, 244]. False dependencies may occur due to the unavailability of physical address information. The ROB has to address these false dependencies to avoid corrupting data and computation. The occurrence of false dependencies and their resolution depends on the actual implementation of the memory subsystem. Intel also uses a proprietary technique for *memory disambiguation* and *dependency resolution logic* in the CPUs to predict and resolve false dependencies at an earlier pipeline stage.

L1 Cache Bottlenecks. The L1 cache port has a limited bandwidth, and simultaneous accesses will block each other. Older CPU generations adopted multiple banks as a workaround to this problem [107], in which each bank can operate independently and serve one request at a time. While the multi-bank workaround partially solves the bandwidth limit, it creates the cache-bank conflict phenomena where simultaneous access to the same bank creates resource contention and delay. Intel has resolved the cache bank conflict issue with the Haswell generation [163]. A potential solution is to use multi-port banks to avoid bank conflicts.

Mentioned false dependencies are the source of another potential bottleneck for accessing the same cache offset [107, 163]. Simultaneously load and store operations with addresses, which are 4 kB away from each other, are problematic, and they halt each other. The CPU cannot determine the dependency from the virtual address, and addresses with the same last 12 bits have the chance to map to the same physical address. Such simultaneous access can happen between two sibling threads within the same CPU core or during the out-of-order execution. There is a chance that a memory store/read might be dependent on a `load/store` with the same last 12 bits of the address. A microarchitecture may not determine such dependencies on the fly; thus, they cause latency.

Transactional memory. Hardware transactional memory (HTM) allows the atomic execution of memory operations. Exceptions during an HTM transaction abort the entire transaction and discard the outcome.

Intel Transactional Synchronization Extension (TSX) implements HTM by introducing a new set of barrier instructions in which application developers can define a block of code to be executed atomically by surrounding it with the `xbegin` and `xend` instructions. The CPU only commits the results of a transaction if the entire block executes successfully. Cache conflicts, hazards within the memory subsystem, and interrupts, which may affect the atomicity operations, abort TSX transactions. Conflicting cache and memory accesses may have been introduced by another sibling thread or by the same thread. Intel TSX has been abused for both microarchitectural attack and defense [129, 188, 299, 302, 311]. Local adversaries can also benefit from the TSX extension to silently and efficiently suppress architectural faults.

2.1.3 Multithreading

Simultaneous multithreading. Modern CPU designs promote resource sharing via another optimization technique called simultaneous multithreading (SMT) [344]. Even with all the previously mentioned optimizations, the program logic adheres to dependencies, blocking the pipeline from using its full potential. SMT allows multiple threads to execute on the same core simultaneously while sharing the same resources. As a result, if one resource is busy by a thread, other threads can consume the remaining available resources. SMT allows the CPU to use each physical CPU core as multiple virtual logical CPUs. The software stack sees each logical CPU as a separate CPU, and each logical CPU can execute a different OS thread simultaneously. According to memory protection semantics, these threads are architecturally isolated from each other and only access their intended data.

Intel Hyperthreading and AMD SMT support two concurrent threads per core or logical CPUs. On Intel CPUs, these logical CPUs share some of the resources, e.g., store buffer, in a compartmentalized fashion. It statically divides such a resource into two separate sections upon activation of the second thread. Other resources, such as the fill buffer, are time shared. While each competes for the same resources within the core, two sibling threads can belong to independent security domains. For Intel, hyper-threading is a critical optimization feature that can provide modern servers with up to 30% performance gain with the same CPU die size. However, Intel Hyperthreading has suffered from various microarchitectural side channels, since the CPU has to synchronously share resources such as translation lookaside buffer (TLB) [123] and execution ports [15]. ZombieLoad [300] and RIDL [356] showed data leakage due to the sharing of the fill buffer across sibling CPU threads.

Scheduling & multitasking. The operating system manages another level of resource sharing to maximize both DRAM and CPU usage further. The operating system can periodically schedule different processes on different CPU cores (logical CPUs) [230, 291]. As a result, different processes share CPU resources, and each logical CPU executes instructions from one task at a time and switches to another one. For core-private resources, such asynchronous sharing of resources has less chance of creating contention across separate security domains than SMT. Depending on the pipeline's depth and timing of difference operations, contention across separate security domains is still possible during a context switch.

2.2 Microarchitectural Side Channels

More than a decade ago, several researchers introduced the first microarchitectural side-channel attacks, known as cache attacks [32, 267, 269]. In Section 2.2.1, we discuss cache attacks. Section 2.2.2 generalizes such contention-based side-channel attacks by discussing information leakage due to other shared microarchitectural resources. Section 2.2.4 discusses how *rowhammer* additionally compromises integrity of applications and data. Finally, in Section 2.2.3, we go over some of the applications of these attacks on compromising secrets and relaxing security guarantees.

2.2.1 Cache Attacks

Cache attacks create an unconventional information channel between two separate security domains by exploiting CPU caches shared across these domains [182, 223, 265]. Adversaries can exploit cache attacks where they share system cache with benign users. In scenarios where adversaries can co-locate with a victim on the same CPU core, they can attack core-private resources such as L1 cache, e.g., OS adversaries [245]. In some platform-as-a-service (PaaS) cloud environment, virtualization platforms may allow sharing of logical CPUs of a core to different VMs; however, attacks on the shared LLC have a higher impact since the CPU cores share the LLC. In cache timing attacks, the attacker either measure the timing of the victim operations, e.g., *Evict+Time* [267] or the timing of his own memory accesses, e.g., *Prime+Probe* [182], *Flush+Reload* [384].

In a cache timing attack, the attacker generally needs to access an accurate time resource such as `rdtsc` instruction. The `rdtsc` instruction reads the current value of the time stamp counter (TSC), which is equivalent to the number of cycles since the CPU resets. Since the CPU runs with a very high frequency, TSC serves as a very high-precision timer. In platforms that such an instruction is not accessible, it is still possible to craft high-resolution timers [301], e.g., by using a shared counter [223]. Note that an architectural interface influenced by the cache's behavior like the TSX can enable cache attacks that do not require timers [91].

Resolution of cache attacks. The spatial resolution of cache attacks depends on a cache line's size, *i.e.*, 64 bytes on most systems. This spatial resolution implies that cache attacks can learn how victims' memory access pattern spans across 64-byte granularity. Note that memory-access patterns may belong to either data or code pages, as both are frequently wang2017cached. In the former, the attacker can learn secret-dependent memory lookups [182], while in the latter, the attacker can learn about secret-dependent branches [226], at runtime. In the basic form, attacks perform a few observations per an entire victim operation. The temporal resolution of an attack defines how often an adversary can observe the victim's access pattern. In specific scenarios, adversaries can improve these attacks' temporal resolution by interrupting the victim and collecting information about the intermediate memory states [133].

Flush+Reload. We now discuss some of the standard exploitation techniques that attackers can use to exploit the cache as a side channel. Flush+Reload is a cache-attack technique that exploits the difference in memory-access times for wang2017cached and unwang2017cached shared memory addresses [384]. In a Flush+Reload attack, the attacker flushes the cache line for a shared memory address using the `clflush` instruction and, after a few cycles, measures the time for accessing this target address. In the meantime, if nobody accesses this address, the CPU does not cache the data, so the attacker will observe that the access time is high. However, if another execution context accesses the address, the CPU caches the data, and the attacker will observe a fast access time.

Based on the Flush+Reload technique, Researchers have demonstrated attacks to steal cryptographic keys [31, 130, 133, 186, 270] as well as to spy on user's behavior [130]. In practice, Flush+Reload is only applicable to environments where the victim and attacker share memory pages. Some cloud environments may support the sharing of read-only code pages across different VMs to optimize memory usage [309]. As the `clflush` instruction follows the same protection rules as memory load operations, attackers can use this technique to spy on these shared code pages, e.g., tracking the runtime decision of a particular branch instruction when accessing different cache lines. More recently, transient execution attacks use Flush+Reload as covert channel [207, 224, 300]. Similarly, in Chapter 3, we use the Flush+Reload to construct a covert channel from the transient to the architectural domain [247].

Prime+Probe attack. Prime+Probe is another cache-attack technique that is more flexible, as it only needs access to shared caches but does not require access to shared memory pages. In the Prime+Probe attack, the attacker first fills an entire cache set by accessing memory addresses that point to the same cache set, the *eviction set*. Later, the attacker checks whether the victim program has displaced any entry in the cache set by reaccessing the eviction set and measuring the execution time. If this is the case, the attacker can detect congruent addresses since the displaced entries cause increased access time.

Prime+Probe is widely applicable to various threat models and attack scenarios. However, the attacker requires finding an eviction set, which is more difficult for the LLC due to the translation of virtual addresses to physical addresses. Note that the CPU maps the data to different cache sets of the LLC using the physical address. Previous attacks have used software interfaces such as hugepages [155] or the virtual-to-physical page mapping in the pagemap file of Linux [223] to overcome this challenge. Nowadays, most OSes restrict user-level applications to access these interfaces due to the associated security risk. As a result, we later show how leakage of the physical page mapping from the microarchitecture can boost the Prime+Probe attack [187].

2.2.2 Generalization to other Shared Resources

Researchers have applied techniques from cache attacks to other microarchitectural buffers that behave like a cache, e.g., branch predictors [6, 8, 104] and TLB [123]. Similarly, adversaries who share these resources with security-critical benign applications can learn about the memory-access and execution pattern of victims through timing the usage of these resources [115]. In general, memory components such as cache and internal buffers are not the only microarchitectural attack surfaces. Adversaries can exploit other microarchitectural components such as execution ports [15] and execution units e.g., floating-point unit [21] to construct side channels.

With the wide-spread use of SMT, core-private side-channel attacks have gained more attention, as attacks such as Portsmash [15] are only feasible due to real-time sharing of core-private resources, which otherwise were not practical. Side-channel attacks exploiting cache bank conflicts also rely on synchronous resource contention [385]. In a cache bank conflict attack like the *CacheBleed* [385], the adversary repeatedly performs simultaneous reads, which are enabled thanks to SMT, to the same cache bank and measures their completion time. A victim on a co-located sibling CPU thread who access the same cache bank cause latency to the attacker's memory reads. Toward the same direction, in Section 2.3, we discuss our contribution of an attack that stems from simultaneous access to addressing logics within the CPU core.

2.2.3 Side-channel Leakage in Practice

Memory-access and execution patterns leak information about data-dependent memory operations and control flow decisions at runtime. These patterns are used to steal secrets such as cryptographic keys [197, 270], or user's password [222], to bypass software protections such as Address Space Layout Randomization (ASLR) [103], or to spy on other users [136]. We discuss some of these use cases in more detail.

Cryptographic leakage. Side-channel leakage on cryptographic implementations occurs because of secret-dependent operations with visible footprints. Researchers have demonstrated side-channel cryptanalysis in both network and local adversarial scenarios [54, 228, 267]. As mentioned, a typical model for exploiting microarchitectural side channels is for a spy process to cause resource contention with a victim process and to measure the timing of its own or the victim operations [182, 281, 340]. The observed timing behavior leaks critical runtime metadata from cryptographic subroutines. Among the shared resources, cache attacks have received significant attention, and researchers have demonstrated their practicality in scenarios such as cloud computing [128, 155, 182, 281, 384, 390]. A distinguishable feature of cache attacks is tracking memory accesses with high temporal and spatial resolution. Thus, they excel at exploiting cryptographic implementations with secret dependent memory accesses [31, 154, 267, 343]. Examples of such vulnerable implementations include using S-Box tables [364], modular exponentiation, and scalar multiplication based on pre-computed values [143, 206].

Constant-time implementations have virtually eliminated the first generation of side-channel cryptanalysis that exploit apparent secret-dependent timing behavior. The standard view is that the performance penalty is the only downside that is no need to be further worried once paid. However, this is far from reality, and constant-time implementations may give a false sense of security. A commonly overlooked fact is that constant-time implementation and related protections are relative to the underlying hardware [116]. Major obstacles are preventing us from obtaining correct constant-time behavior. CPUs continuously evolve with new microarchitectural features rolled quietly with each new release, and the variety of such subtle features makes comprehensive evaluation impossible. A great example is the cache bank conflicts attack against OpenSSL

RSA scatter-gather implementation [385]: it shows that adversaries with intra-cache-line resolution can successfully bypass constant-time techniques relied on cache-line granularity. Consequently, what might appear as a perfect constant-time implementation becomes insecure in the next CPU release – or worse – an unrecognized behavior might be discovered, invalidating the earlier assumption.

Weakening security. Knowledge of address mappings that are only available to the privileged software weakens software systems' security against microarchitectural side channels. In some threat models, the randomization of virtual addresses by the OS is considered the secret to strengthening software security against architectural vulnerabilities, e.g., buffer overflows. For example, the kernel address space layout (KASLR) mitigation reduces the impact of software-based vulnerabilities within the OS kernel by making it harder to find code gadgets. Microarchitectural attacks can recover virtual address information and break KASLR by exploiting the Translation Lookaside Buffer (TLB) [152], Branch Target Buffer (BTB) [103] and Transactional Synchronization Extensions (TSX) [188]. Additionally, Gruss et al. [126] exploit the timing information obtained from the `prefetch` instruction to leak the physical address information. The main obstacle to this approach is that the `prefetch` instruction is not accessible in JavaScript, and one can disable such instructions in sandboxed environments [386]. In contrast, our technique in Section 2.4 applies to sandboxed environments, including JavaScript.

2.2.4 The Rowhammer Paradigm

Leaky DRAM cells. DRAM cells, responsible for storing bits of data, leak over time, and need to be refreshed periodically to maintain their data. At the microarchitectural level, DRAM consists of multiple memory banks, and each bank has several rows. When the CPU accesses a memory location, the memory controller, with the DRAM's help, activates the corresponding row and loads it into a row buffer. If the CPU reaccesses the same row, it is called a row hit, and the row buffer will serve the request. Otherwise, it is called a row conflict. The previous row will be deactivated and copied back to the original row location, after which the memory controller actives the new row. The memory controller on the DRAM or the CPU has to refresh all the DRAM rows periodically by copying them to and from the row buffer. If the refresh cycle fails to refresh the victim row fast enough, that leads to bit flips, *i.e.*, memory errors.

Rowhammer. The *rowhammer* attack [202] repeatedly causes cells within a victim row to leak faster by activating the neighboring rows under the control of the attacker. Researchers have demonstrated that neighboring rows' electromagnetic effect causes more frequent bit flips before the refresh cycle mitigates such an error. Some memory cells are more susceptible to this phenomenon due to inconsistencies introduced during manufacturing. Once attackers locate cells that suffer from random bit flips, they can exploit it by tricking victims into placing a security-critical data structure or code page at that particular location and triggering the bit flip again [125, 304, 377].

The rowhammer attack requires fast access to the same DRAM row through bypassing the CPU cache, e.g., using `clflush` [202]. Additionally, cache eviction based on an eviction set can also result in access to DRAM cells when `clflush` instruction is not available [25, 127]. Efficiently building eviction sets may thus also enhance rowhammer attacks. It is essential for a successful rowhammer attack to collocate multiple memory pages within the same bank and adjacent. The memory controller uses several bits of the physical address, depending on the hardware configuration, to map memory pages to banks [273]. Since the rows are generally placed sequentially within the banks, access to adjacent rows within the same bank will become relatively more straightforward for the attackers if they have access to contiguous physical pages.

Insufficient mitigation for security. Although several countermeasures exist to mitigate the impact of errors for DRAM cells, they often fail to prevent intentional bit flips that are initiated by a determined attacker. One approach would be to reduce the refresh cycle to decrease the chance of bit flips. Reducing the refresh cycle comes with extra overhead, as the memory controller has to spend more time on refresh commands, which blocks regular memory operations. Besides, this approach does not even provide strong protection against rowhammer [201, 367]. Some DRAMs support error-correcting codes (ECC), but the ECC on DRAMs can only detect and fix up to 2 or 3 bits errors due to performance constraints. Researchers have shown that rowhammer is still possible on DRAMS with ECC [75], and failures due to the ECC checks provide an oracle for attackers to leak data bits [216]. Another widely-deployed mitigation, target-row refresh (TRR), aims to adaptively refresh rows adjacent to accessed rows to reduce the chance of intentional bit flips. However, *TRRespass* [111] confirms that this ad-hoc approach is insufficient to protect against these attacks fully.

2.3 MemJam Attack on Virtual Address Aliasing

Contribution of MemJam. As we have mentioned, constant-time implementation techniques have become an indispensable tool in fighting microarchitectural side-channel attacks. These techniques engineer the memory accesses of cryptographic operations to follow a uniform key independent pattern. However, constant-time behavior is dependent on the underlying architecture, which can be highly complex and often incorporates unpublished features. The *CacheBleed* attack is based on cache-bank conflicts. Therefore, it invalidates that microarchitectural side channels can only observe memory with cache-line granularity [385]. In this section, we propose MemJam, which utilizes *4k aliasing* to establish a side-channel attack that exploits memory's false dependency read-after-write events and provides a high-quality intra-cache-line timing channel [246]. We show how to dramatically slow down the victim's access to the specific memory blocks and how attackers can use this read latency to recover low address bits of the victim's memory accesses. Compared to *CacheBleed*, which is limited to older CPU generations, MemJam is the first intra-cache-line attack applicable to all significant Intel CPUs, including the latest generations, and also applies to the SGX environment. In Section 6.1, we use MemJam to demonstrate key recovery against three different cryptographic implementations that are resistant against cache attacks.

2.3.1 False Dependencies due to 4k Aliasing

MemJam uses *false dependencies*. Data dependency occurs when an instruction refers to the data of a preceding one. In pipelined designs, hazards and pipeline stalls can occur from dependencies if the previous instruction has not finished. There are cases where false dependencies occur, *i.e.*, the pipeline stalls even though there is no true dependency. False dependencies could occur for several reasons, including, but not limited to, register reuse, and limited address space for the execution units. False dependencies degrade instruction-level parallelism and cause overhead. Modern CPUs eliminates false dependencies arising from register reuse by a register renaming approach. However, like Intel, CPU manufacturers provide optimization guidelines for the software and compiler developers to address some of these false dependencies with software workarounds [163, 164], e.g., *partial register stalls*.

4k aliasing. In this contribution, we focus on a critical false dependency described as *4k aliasing*. The CPU may see data blocks multiplying 4k apart in the address space as a dependent. 4k aliasing happens due to virtual addressing of L1 cache, where data is accessed using virtual addresses but tagged and stored using physical addresses. Multiple virtual addresses can refer to the same data with the same physical address, and the ultimate determination of dependency for concurrent memory accesses requires virtual address translation. Physical and virtual addresses share the last 12 bits, and any data accesses whose addresses differ in the last 12 bits (*i.e.*, the distance is not 4k) cannot have a dependency. For the reasonably rare remaining cases, address translation needs to be done before resolving the dependency, which causes latency. Note that dependencies can occur at the *word* or cache *line* granularity (*i.e.*, ignoring the last 2, or last 6 bits of the address, respectively). We exploit these rare false dependencies due to 4k aliasing to spy on the memory. Attackers can deliberately process falsely dependent memory accesses by matching their address's last 12 bits with a security-critical address inside a victim process.

Multiple references mention *4k aliasing* as an optimization problem existing on all major Intel CPUs [107, 163]. We verify the results of Yarom et al. [385], the only security-related work regarding false dependencies, which mentioned *write-after-read* dependencies. The resulting timing leakage by store stall after a load is not sufficient to be used in any practical attack, e.g., stealing cryptographic keys. MemJam exploits a different channel due to the false dependency of *read-after-write*, which causes a higher latency and is thus merely observable. Intel Optimization Manual highlights the *read-after-write* performance overhead in various sections [163]. As described in Section 11.8 of this document, this hazard occurs when a memory load closely follows a memory store. It causes the pipeline to reissue the load operation with a potentially five cycles penalty. In contrast, to load bounds, in Section B.1.4 on memory bounds, the top-down microarchitectural analysis method (TMAM) reports store bounds as a fraction of cycles with low execution port utilization and small performance impact. This top-down characterization is a hierarchical organization of event-based metrics that identify the dominant performance bottlenecks in an application. These descriptions in various sections highlight that *read-after-write* stall is considered more critical than *write-after-read* stall.

2.3.2 Memory Dependency Fuzz Testing

We performed experiments to evaluate the memory dependency behavior between two sibling CPU threads within a core. In these experiments, we have thread $\mathcal{A}$ and $\mathcal{B}$ running on the *same* physical core, but as *different* CPU threads, as shown in Figure 2.4. Both threads perform memory operations; only thread $\mathcal{B}$ measures its timing and hence the timing impact of introduced false dependencies.

Experimental setup and assumptions. Our experimental setup is a Dell XPS 8920 desktop machine with an Intel Core i7-7700 CPU running Ubuntu 16.04. The Core i7-7700 has four hyper-threaded physical cores. Our only assumptions are that the attacker can co-locate on one of the CPU pairs (CPU threads) within the same physical core as the victim.

Read-after-read (RaR). In the first experiment, the two logical threads $\mathcal{A}$ and $\mathcal{B}$ read from the same shared cache and can potentially block each other. This experiment

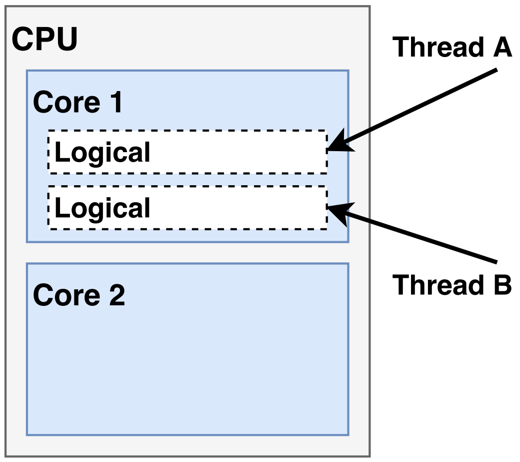


Figure 2.4: Thread A and B both run on the same core, and introduce and probe stall hazards.

```
loop:
rdtscp;
mov %eax, (%r9);
movb 0x0000(%r10), %al;
movb 0x1000(%r10), %al;
movb 0x2000(%r10), %al;
movb 0x3000(%r10), %al;
movb 0x4000(%r10), %al;
movb 0x5000(%r10), %al;
movb 0x6000(%r10), %al;
movb 0x7000(%r10), %al;
add $4, %r9;
dec %r11;
jnz loop;
```

Listing 2.1: Probe Reads

```
loop:
rdtscp
mov %eax, (%r9);
movb %al, 0x0000(%r10);
movb %al, 0x1000(%r10);
movb %al, 0x2000(%r10);
movb %al, 0x3000(%r10);
movb %al, 0x4000(%r10);
movb %al, 0x5000(%r10);
movb %al, 0x6000(%r10);
movb %al, 0x7000(%r10);
add $4, %r9
dec %r11
jnz loop
```

Listing 2.2: Probe Writes

Figure 2.5: Listings 1 and 2 are used to probe 8 parallel reads and writes, respectively. *r9* points to a measurement buffer, and *r11* is initialized with the probe count.

can reveal cache bank conflicts, as used by *CacheBleed* [385]. $\mathcal{B}$ uses Listing 2.1 to perform read measurements, and $\mathcal{A}$ repeatedly reads from different memory offsets and tries to introduce conflicts. $\mathcal{A}$ reads from three different type of offsets: **(1)** Different cache line than $\mathcal{B}$, **(2)** same cache line, but different offset than $\mathcal{B}$, and **(3)** same cache line and same offset as $\mathcal{B}$. The results in Figure 2.6 show that we can not observe any difference between the histograms for three cases. This experiment verifies the lack

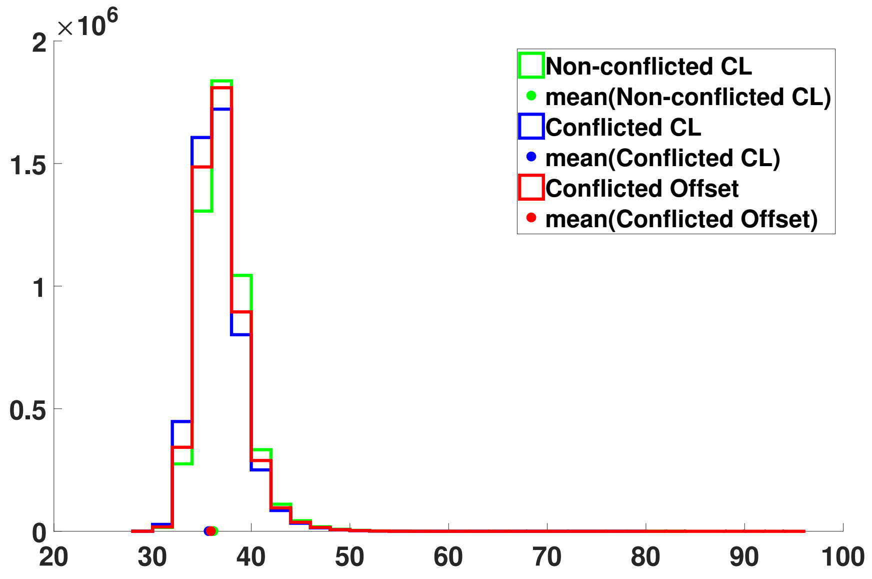


Figure 2.6: Three different scenarios where the two CPU threads perfrom memory loads to different cache lines (green), same cache line (blue) and same offset (red). The three histograms are almost indistinguishable.

of cache bank conflicts on the 7th generation Intel CPUs, as we would otherwise have observed simultaneous loads to the same cache offset to be distinguishable from the other two cases.

Write-after-read (WaR). In this experiment, thread $\mathcal{A}$ constantly loads from different type of memory offsets, while thread $\mathcal{B}$ uses Listing 2.2 to perform measurements of store operations. Figure 2.7 shows the three histograms for the second experiment on the potential false dependency of write-after-read. The standard deviation for conflicted cache line (blue) and conflicted offset (red) between thread $\mathcal{A}$ and $\mathcal{B}$ is distinguishable from the green bar where there is no cache line conflict. This observation shows a high capacity cache granular behavior. Still, the slight difference between conflicted line (blue) and offset (red) verifies the previous results stating a weak leakage due to offset dependency [385].

Read-after-write (RaW). Figure 2.8 shows an experiment on measuring false dependency of read-after-write, in which thread $\mathcal{A}$ frequently stores to different memory offsets. Thread $\mathcal{B}$ uses Listing 2.1 to perform measurements of load operations. Accesses to three different types of offsets are distinguishable. The conflicted cache line accesses (blue) are distinguishable from non-conflicted accesses (green). More importantly, conflicting accesses to the same offset (red) are also distinguishable from conflicted cache line accesses.

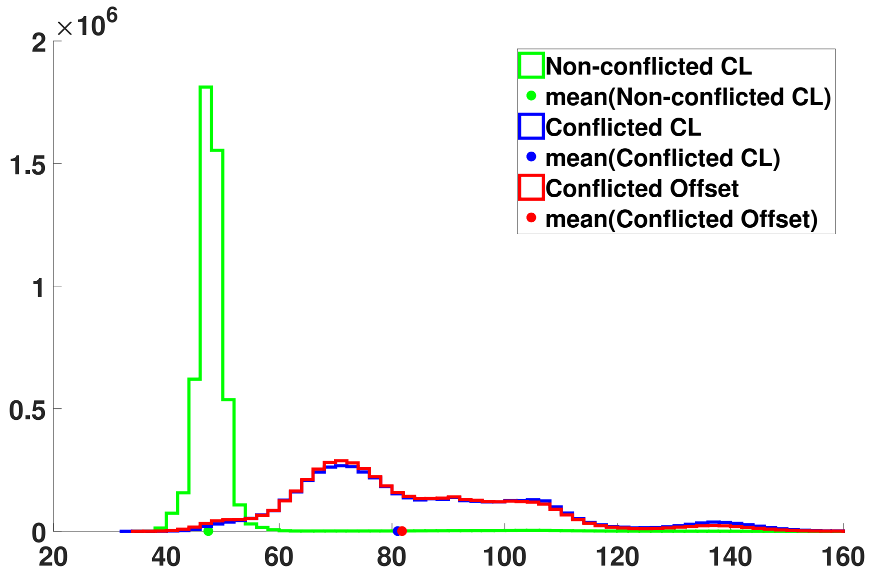


Figure 2.7: Similar to Figure 2.6, three different offsets are tested. We can easily distinguish cache-granular accesses when testing write-after-read dependencies, but detecting intra-cache-line leakage is not practical using this strategy.

We exploit this timing behavior for introducing a side channel with an **intra-cache-line** granularity. There is an average of 2 cycle penalty if we access the same cache set and a ten cycles penalty if we access the same offset. Note that we separately verified that the word offsets in our platform have 4-bytes granularity. From an adversarial standpoint, this means that an adversary learns about bits 2-11 of the victim's memory access, in which 4 bits (bits 2-5) are intra-cache-line information, and thus goes beyond any other microarchitectural side channels that are known to exist on 6th and 7th generation Intel CPUs. We later discuss how we can exploit this extra information (in Figure 2.9), for demonstrating practical attacks.

Read-after-weak-Write (RawW). In this follow-up experiment on the read-after-write conflicts, we tried a less greedy strategy on the conflicting thread. Rather than continually performing store operations to the same offset, $\mathcal{A}$ stores to the same memory offset with some gaps filled with other memory accesses and instructions. Figure 2.10 shows that the channel dramatically becomes less effective. This observation tells us that the load penalty will be more effective with frequently performing stores to the same offset without additional instruction between the memory load operations. In this regard, we will use Listing 2.3 in our attack to achieve the maximum contentions.

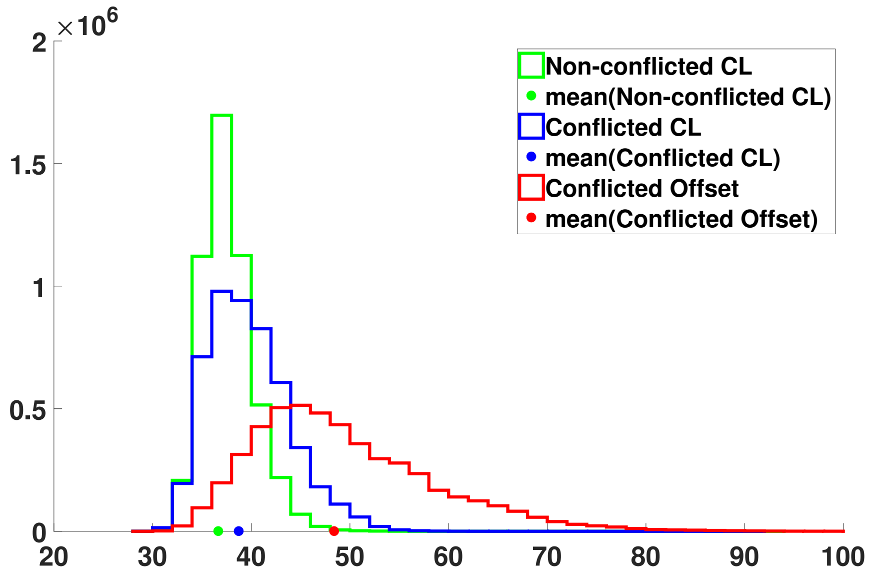


Figure 2.8: RaW latency has distinguishable characteristics for the conflicted word offset (red).

Read-after-Write latency. In the last experiment, we tested the delay of execution over various conflicting memory loads. We created a code stub with 64 memory load instructions and a random combination of instructions between memory loads to make a more realistic computation. We chose the combination to avoid unexpected halts and maintain the parallelism of all load operations. We measure this computation's execution time on $\mathcal{B}$, while $\mathcal{A}$ is storing to a conflicting offset. First, we measured the computation with 64 memory loads to separate addresses without conflicts. Our randomly generated code stub takes an average of 210 cycles to execute. On each step of the experiments, as shown in Figure 2.11, we change some of the memory offsets to have the same last 12 bits of address as the conflicting offset of store operations on $\mathcal{A}$.

We observe growth in the latency for memory loads by increasing the number of conflicting loads. Figure 2.11 shows the results for several experiments. In all of them, the overall execution time of $\mathcal{B}$ is strongly dependent on the number of conflicting memory loads. Hence, we can use the RaW dependency to introduce assertive timing behavior using bits 2-11 of a chosen target memory address.

2.3.3 Leaking with Intra-Cache-Line Resolution

MemJam exploits read-after-write false dependencies to introduce timing behavior to otherwise constant-time implementations. An attacker who targets a cryptographic

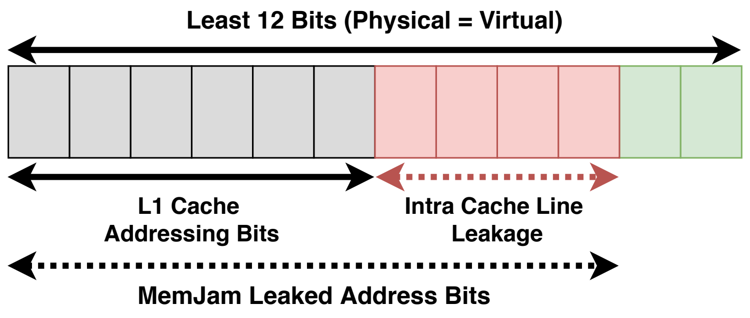

Least 12 Bits (Physical = Virtual)
L1 Cache Addressing Bits
Intra Cache Line Leakage
MemJam Leaked Address Bits

Figure 2.9: Intra-cache-line Leakage: MemJam latency is related to 10 address bits, in which 4 bits are intra cache level bits.

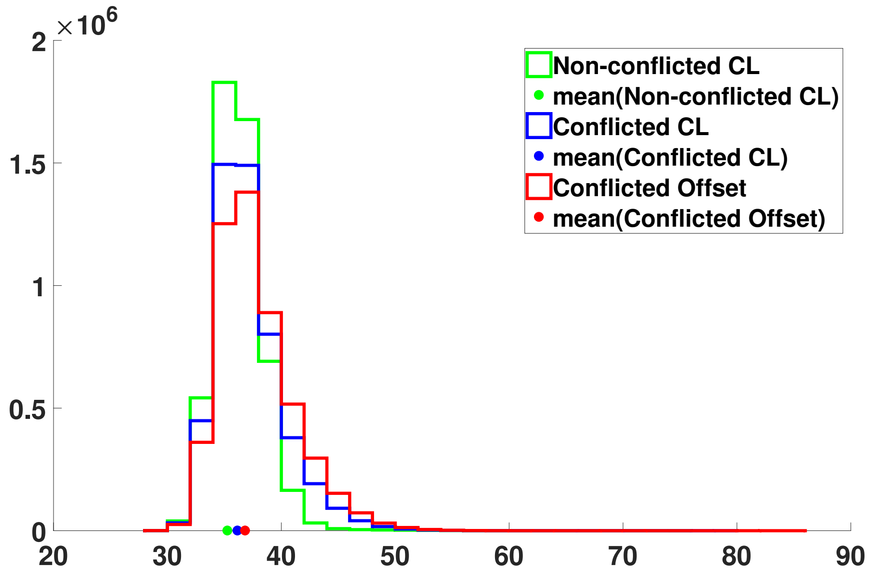


Figure 2.10: RawW: Compared to Figure 2.8, this shows a lower impact on access latency.

operation can use the resulting latency for a correlation attack. MemJam proceeds with the following steps:

1. attacker launches a process constantly writing to an address using Listing 2.3 where the last 12 bits match the virtual memory offset of a *critical* data that is loaded by the victim's process.

2. While the attacker's conflicting process is running, the attacker queries the victim to perform a security-critical operation like encryption and records the output ciphertext and execution time pair of the victim. Higher time means more access to that *critical* offset.

3. attacker repeats the previous step, collecting ciphertext and time pairs.

```
mov %[target], %rax;
write_loop:
    .rept 100;
    movb $0, (%rax);
    .endr;
jmp write_loop;
```

Listing 2.3: Write Conflict Loop: Unnecessarily instructions are avoided to minimize usage of other CPU units and to maximize the RaW conflict effect.

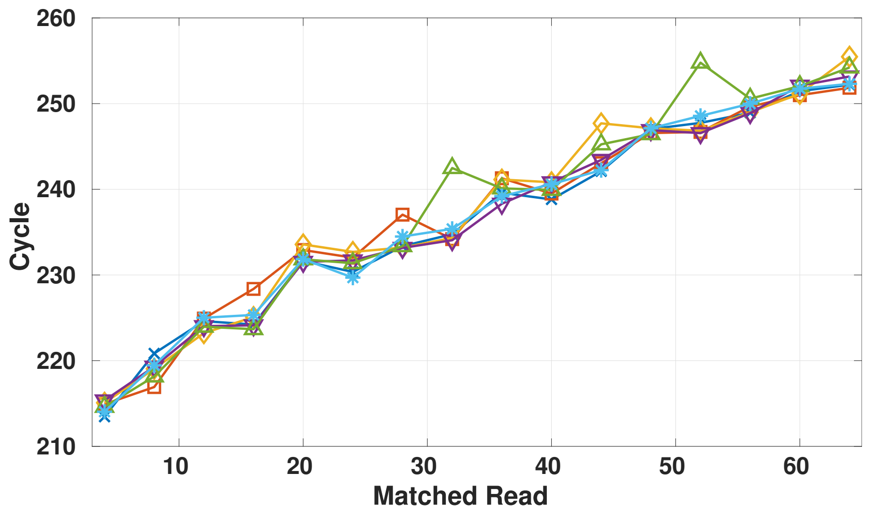


Figure 2.11: The cycle count for mixed operations with RaW conflicts. More conflicts cause higher delay.

Our attack methodology resembles the *Evict+Time* strategy proposed initially by Tromer et al. [340], except that the attacker uses false dependencies due to 4K aliasing rather than evictions to slow down the target *and* that the slowdown only applies to a 4-byte block of a cache line. Furthermore, *all* of the victim's accesses addresses with the same last 12 bits are slowed down while a cache eviction only slows the first memory accesses.

Based on the intra-cache-line leakage in Figure 2.9, we can divide a 64-byte cache line into 4-byte blocks and hypothesize that we can correlate the access counts to the first one with the running time of the victim. Simultaneously, the attacker jams memory loads to that (chosen first) block, *i.e.*, the attacker expects to observe a higher time when there are more accesses by the victim to the targeted 4-byte block and a lower time when there is a lower number of accesses. Based on this hypothesis, we can apply a classical correlation-based side-channel approach [208] to attack implementations of different block ciphers, namely 3-DES, AES, and SM4. We will extensively discuss these cryptanalysis results in Section 6.1.

Applicability of MemJam. MemJam exploits false dependencies of memory read-after-writes across sibling CPU threads and turns this primitive into a technique similar to cache-based timing attacks, but with a 4-byte spatial resolution. Consequently, countermeasures aimed at providing uniform accesses at cache-line granularity do not protect against MemJam. For MemJam to work, the false dependencies need to impact the memory load operations after a false conflicting store. Section 2.3.3 highlights the availability of the cache bank conflicts and the 4k aliasing leakage source: While bank conflicts are limited to few CPU generations, 4k aliasing is present in all Intel CPUs released in the last ten years. Thus, MemJam applies to all Intel CPUs that support SMT, e.g., Intel hyperthreading. In Section 6.1, we show how MemJam enables novel cryptanalysis of several encryption schemes.

Release	Family	Cache Bank Conflicts	4k aliasing
2006	Core	✓	✓
2008	Nehalem	×	✓
2011	Sandy bridge	✓	✓
2013	Silvermont, Haswell, Broadwell	×	✓
2015	Skylake	×	✓
2016	KabyLake	×	✓

Table 2.1: Intel CPU families and availability of the leakage channels. Major Intel CPUs suffer from 4k aliasing and are vulnerable to MemJam [107].

2.4 Beyond MemJam: Physical Address Aliasing

In *MemJam* [244], we verified a false dependency hazard within the memory order buffer, in which memory operations on non-colliding physical addresses that share the same least 12 significant bits affect each other. We exploit this behavior from separate CPU threads within a core to construct a microarchitectural timing side channel. Sullivan et al. [328] demonstrate a covert channel based on the same 4k aliasing behavior. The authors conclude that an address aliasing check is a two-stage approach: Firstly, it uses page offset for the initial guess. Secondly, it performs the final resolution based on the exact physical address. However, complex memory subsystems may introduce undocumented false dependency hazards.

Contribution of Spoiler. Contrary to public knowledge, we discover that the undocumented *address resolution* logic performs additional partial address checks that lead to a strange but observable aliasing behavior based on the physical address. This undocumented behavior is independent of the aforementioned 4k aliasing behavior [107, 244, 328, 385]. The discovered false dependency happens during the 1 MB aliasing on the physical address of speculative memory accesses.

As discussed in Section 2.1, *Speculative loads* and *store forwarding* are techniques to improve the memory bottleneck in a pipelined out-of-order CPU. The CPU executes the `load` speculatively and forwards the data of a preceding `store` to the `load` if there is a dependency. This enhances performance since the `load` does not have to wait for preceding `stores` to complete. However, the dependency prediction relies on partial address information, which may lead to false dependencies. we observe that failure to resolve false dependencies promptly creates stall hazards, which allows an attacker to construct a new timing side channel.

In this work, we are the first to exploit the dependency resolution logic during *speculative loads* as a timing channel to gain physical address information. We propose the SPOILER attack, which exploits this channel to gain information about the physical page number's eight least significant bits. Then, we discuss how SPOILER improves the *Prime+Probe* cache-attack technique and *rowhammer* attacks. Further, we discuss the possibility of single-threaded *MemJam* attacks where the attacker tracks a victim's memory load after a context switch.

2.4.1 Speculative Load Hazards

Memory `loads` are executed out-of-order, and they may execute before preceding memory `stores`. If one of the preceding `stores` modifies the content of a location in memory, the memory `load` address is referring to, out-of-order execution of the `load` will operate on stale data, which corrupts the program's state. This out-of-order execution of the memory `load` is a speculative behavior since there is no guarantee during the execution time of the `load` that the virtual addresses corresponding to the memory `stores` do not conflict with the `load` address after translation to physical addresses.

Figure 2.12 demonstrates this effect on a hypothetical CPU with 7 pipeline stages. As multiple `stores` may be blocked due to limited resources, the execution of the `load` and dependent instructions in the pipeline, the *load block*, will bypass the `stores` since the MOB assumes the load block to be independent of the `stores`. This speculative behavior improves the memory bottleneck by letting other instructions continue their execution. However, suppose the dependency of the `load` and preceding `stores` is not verified. In that case, the load block may consume incorrect data, which is either falsely forwarded by

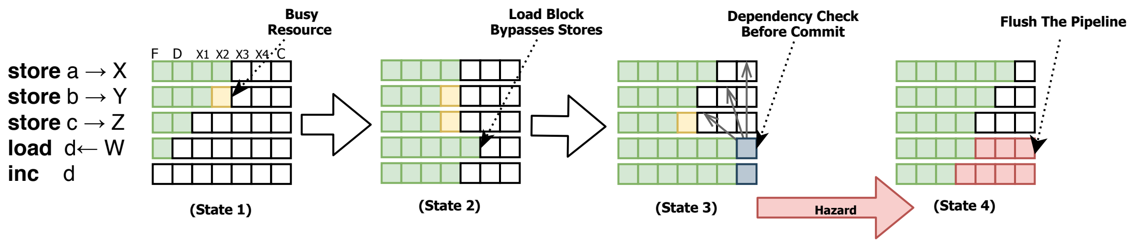


Figure 2.12: The speculative load is demonstrated on a hypothetical CPU with 7 pipeline stages: F = Fetch, D = Decode, X_{1-4} = Executions, and C = Commit. When the memory `stores` are blocked competing for resources (State 1), the `load` will bypass the `stores` (State 2). The load block including the dependent instructions will not be committed until the dependency of the address W versus X,Y,Z are resolved (State 3). In case of a dependency hazard (State 4), the pipeline is flushed and the `load` is restarted.

store forwarding (false dependency) or loaded from a stale cache line (a true unresolved dependency). If the CPU detects a false dependency before committing the `load`, it has to flush the pipeline and re-execute the load block. This flushing and reissuing of instructions will cause observable performance penalties and potentially a timing behavior detectable by an attacker.

Dependency resolution. Dependency checks and resolution occur in multiple stages depending on the availability of the address information in the store buffer. A `load` instruction needs to be checked against all preceding `stores` in the store buffer to avoid false dependencies and ensure the correctness of the data. A potential design in Intel patents [146, 212] suggests three stages for the dependency check and resolution, as shown in Figure 2.13. Note that the exact implementation of the MOB used in Intel CPUs is unpublished, and therefore we cannot be sure about the precise architecture. However, our results agree with some of the possible design choices described in these Intel patents. The three stages are as the following:

1. **Loosenet**: The first stage is the *loosenet* check where the page offsets of the `load` and `stores` are compared. In the case of a loosenet hit, the compared `load` and `store` may be dependent, and the CPU will proceed to the next check stage.

2. **Finenet**: The next stage, called *finenet*, uses upper address bits. 5 The *finenet* can be implemented to check the upper virtual address bits [146], or the physical address tag [212]. Either way, it is an intermediate stage, and it is not the final dependency resolution. In the case of a *finenet* hit, the CPU blocks the `load` and forwards the store data; otherwise, the dependency resolution will go into the final stage.

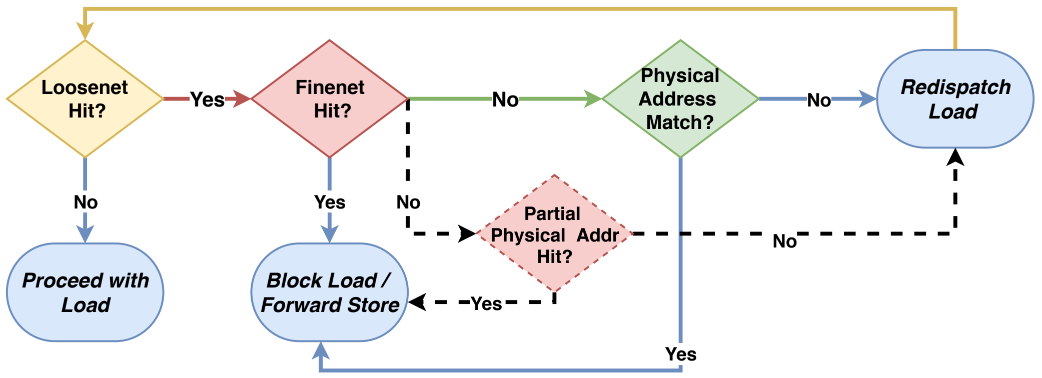


Figure 2.13: The dependency check logic: *loosenet* initially checks the least 12 significant bits (page offset) and the *finenet* checks the upper address bits, related to the page number. The final dependency using the physical address matching might still fail due to partial physical address checks.

3. **Physical Address Matching**: At the final stage, the addressing logic will check the physical addresses. Since this stage is the final chance to resolve potential false dependencies, we expect the addressing logic to check the full physical address. However, one possible design suggests that if the physical addresses are not available, the physical address matching returns true and continues with the store forwarding [146].

Since the page offset is identical between the virtual and physical address, a potential design can perform the loosenet check as soon as the `store` is decoded. Intel optimization manual [163] and the event `Ld_Blocks_Partial:Address_Alias` from Intel hardware performance counters refer to loosenet as a mechanism that only compare the page offsets. Therefore loosenet checks resemble the same behavior that MemJam exploits.

According to some Intel patents, the store buffer may only hold bits 19 to 12 of the physical address [3]. Although the physical address buffer holds the full translated physical address, it is not clear at which stage this information can be available to the MOB. As a result, the *finenet* may check the partial physical address bits. As we discover later, the dependency resolution logic fails to resolve the dependency at multiple intermediate stages due to the full physical address's unavailability.

2.4.2 The Spoiler Leakage

The attack model for Spoiler is the same as rowhammer and cache attacks, where the attacker executes code on the same underlying hardware as the victim. As described in Section 2.4.1, speculative loads may face other aliasing conditions in addition to the 4k

Algorithm 1 Address Aliasing

```
for $p$ from w to PAGE_COUNT do
    for $i$ from w to $0$ do
        $data \xrightarrow{store} buffer[(p-i) \times PAGE\_SIZE]$
    $t_1 = rdtscp()$
    $data \xleftarrow{load} buffer[x \times PAGE\_SIZE]$
    $t_2 = rdtscp()$
    $measure[p] \leftarrow t_2 - t_1$
return $measure$
```

aliasing due to the partial checks on the higher address bits. To confirm this, we design an experiment to observe a speculative load's timing behavior based on higher address bits.

In this experiment, we propose Algorithm 1 that executes a speculative load after multiple `stores` and further make sure to fill the store buffer with addresses that cause 4k aliasing during the execution of the `load`. Having w as the window size, the algorithm iterates over several different memory pages. For each page, it performs `stores` to that page and all previous w pages within a *window*. Since the store buffer size varies between different CPU generations, we choose a big enough window ($w = 64$) to ensure that the `load` has 4k aliasing with the maximum number of entries in the store buffer and hence maximizing potential conflicts. Following the `stores`, we measure the timing of a `load` operation from a different memory page, as defined by x.

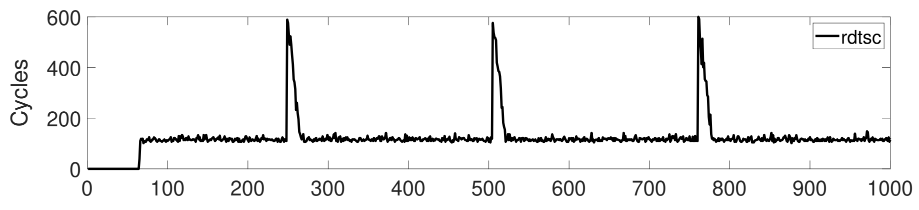
Since we want the `load` to be executed speculatively, we can not use a store fence such as `mfence` before the `load`. As a result, our measurements estimate execution time for the speculative `load` and nearby microarchitectural events. This estimation may include a negligible overhead for the execution of store operations and any delay due to the dependency resolution. If we iterate over a diverse set of addresses with different virtual and physical page numbers but the same page offset, we should monitor discrepancies.

```
rdtscp;
mov %eax, %esi;
mov (%rbx), %eax;
rdtscp;
mfence;
sub %esi, %eax;
```

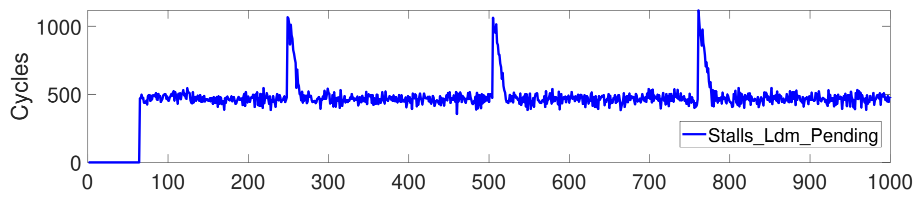
Listing 2.4: Timing measurement of a speculative load.

Speculative dependency analysis. We now use Algorithm 1 and Hardware Performance Counters (HPC) to perform an empirical analysis of the dependency resolution logic. HPCs can keep track of low-level hardware-related events in the CPU. The counters are accessible via special purpose registers and can be used to analyze a program's performance. They provide a powerful tool to detect microarchitectural components that cause bottlenecks. Software libraries such as Performance Application Programming Interface (PAPI) [334] simplifies programming and reading low-level HPC on Intel CPUs.

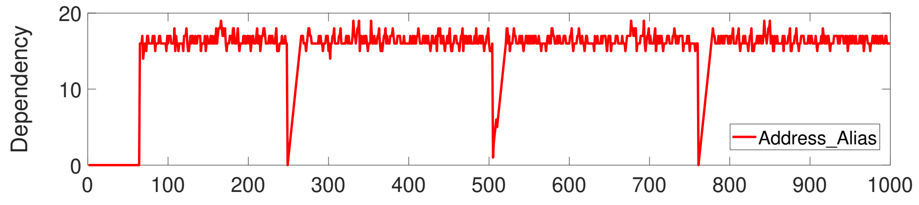
Initially, we execute Algorithm 1 for 1000 different virtual pages. Figure 2.14(a) shows the cycle count for each iteration with a set of 4 kB aliased store addresses. Interestingly, we observe multiple step-wise peaks with very high latency. Then, we use PAPI to monitor 30 different performance counters while running the same experiment. At each iteration, we only monitor one performance counter alongside the timing, as mentioned earlier. After each speculative load, the performance counter value and the `load` time are both recorded. Finally, we obtain the timings and performance counter value pairs, as depicted in Figure 2.14.



(a) Step-wise peaks with a very high latency can be observed on some of the virtual pages



(b) Affected HPC event: `Cycle_Activity:Stalls_Ldm_Pending`



(c) Affected HPC event: `Ld_Blocks_Partial:Address_Alias`

Figure 2.14: SPOILER's timing measurements and hardware performance counters recorded simultaneously.

To find any relation between the observed high latency and a particular event, we compute correlation coefficients between counters and the timing measurements. Since the latency only occurs in the small region of the trace where the timing increases, we only need to compute these regions' correlation. When an increase of at least 200 clock cycles is detected, we use the following s values from timing and the HPC traces to calculate the correlations, where s is the number of steps from Table 2.2 and 200 is the average execution time for a load.

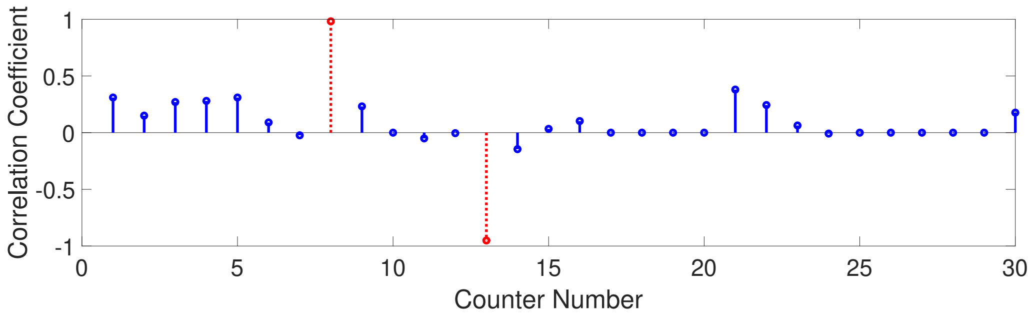


Figure 2.15: Correlation with various HPC events. `Ld_Blocks_Partial:Address_Alias` and `Cycle_Activity:Stalls_Ldm_Pending` (both dotted red) have strong positive and negative correlations, respectively.

As shown in Figure 2.15, two events have a high correlation with the leakage: `Cycle_Activity:Stalls_Ldm_Pending` has the highest correlation of 0.985. This event shows the cycle count when the execution stalls due to a pending `load`. This event `Ld_Blocks_Partial:Address_Alias` counts the number of false dependencies in the MOB when loosenet resolves the 4k aliasing condition and has an inverse correlation with the leakage. Separately, `Exe_Activity:Bound_on_Stores` increases with more number of `stores` within the inner window loop in Algorithm 1, but it does not have a correlation with the leakage. The reason behind this behavior is that the store buffer is full, and additional store operations are pending. However, since there is no correlation with the leakage, we conclude that the timing behavior is not due to the `stores` delay. We also attempt to profile any existing counters related to the memory disambiguation. However, the events `Memory_Disambiguation.Success` and `Memory_Disambiguation.Reset` are not available on the modern architectures that are tested.

Leaking the physical address mapping. In this experiment, we evaluate the observed step-wise latency and its relationship with the physical page numbers by observing the pagemap file. As shown in Figure 2.16, we observe step-wise peaks with a very high latency, which appears once in every 256 pages on average. The 20 least significant bits

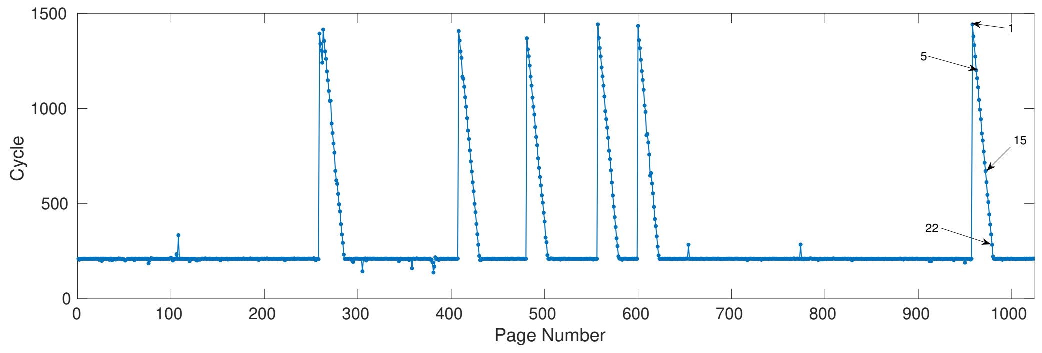


Figure 2.16: Step-wise peaks with 22 steps and a high latency can be observed on some of the pages (*Core i7-8650U* CPU).

of the physical address for the `load` match the physical addresses of the `stores` where we observe high peaks for virtual pages. We always detect peaks with different virtual addresses in our experiments with the same last 20 physical address bits. This observation discovers the existence of 1 MB aliasing effect based on the physical addresses. This 1 MB aliasing leaks information about 8 bits of mapping unknown to the userspace processes.

This observation matches our previous observation of high correlation for the event `Cycle_Activity:Stalls_Ldm_Pending`. Consequently, the speculative load suffers from stalls to resolve the dependency with conflicting store buffer entries after the occurrence of a 1 MB aliased address. This observation verifies that the latency is due to the pending `load`. When the latency is at the highest point, the event that counts 4k aliasings, `Ld_Blocks_Partial:Address_Alias` drops to zero, and it increments at each down step of the peak. This behavior implies that the *loosenet* check does not resolve the rest of the store dependencies whenever there is a 1 MB aliased address in the store buffer.

Further evaluation. In the previous experiment, the execution time of the `load` operation that is delayed by 1 MB aliasing decreases gradually in each iteration (Figure 2.16). The number of steps to reach the expected execution time is consistent on the same CPU. When the first store in the window loop accesses a memory address with the matching 1 MB aliased address; the latency is at its highest point, marked as "1" in Figure 2.16. As the window loop accesses this address later in the loop, it appears closer to the `load` with a lower latency like the steps marked as 5, 15, and 22. This observation matches the *carry chain algorithm* described by Intel [146], where the aliasing check starts from the most recent `store`.

As shown in Table 2.2, experimenting with various CPU generations shows that the number of steps has a linear correlation with the store buffer size, which depends on the microarchitecture. While the leakage exists on all Intel Core CPUs starting from the first

CPU Model	Architecture	Steps	SB Size
Intel Core i7-8650U	Kaby Lake R	22	56
Intel Core i7-7700	Kaby Lake	22	56
Intel Core i5-6440HQ	Skylake	22	56
Intel Xeon E5-2640v3	Haswell	17	42
Intel Xeon E5-2670v2	Ivy Bridge EP	14	36
Intel Core i7-3770	Ivy Bridge	12	36
Intel Core i7-2670QM	Sandy Bridge	12	36
Intel Core i5-2400	Sandy Bridge	12	36
Intel Core i5 650	Nehalem	11	32
Intel Core2Duo T9400	Core	N/A	20
Qualcomm Kryo 280	ARMv8-A	N/A	*
AMD A6-4455M	Bulldozer	N/A	*

Table 2.2: 1 MB aliasing on various architectures: The tested AMD and ARM architectures, and Intel Core generation do not show similar effects. The Store Buffer (SB) sizes are gathered from Intel Manual [163] and *wikichip.org* [369, 370, 371].

generation, the timing effect is higher for the more recent versions with a bigger store buffer size. The analyzed ARM and AMD CPUs do not show similar behavior. Note that we use `rdtscp` for Intel and AMD CPUs and the `clock_gettime` for ARM CPUs to perform the time measurements.

As our time measurement for speculative load suggests, it is impossible to reason whether the high timing is due to a very slow `load` or commitment of store operations. Since the step-wise delay matches the store buffer entries, this delay may be due to the dependency resolution logic performing a pipeline flush and restart of the `load` for each 4 kB aliased entry starting from the 1 MB aliased entry. Another hypothesis would be that the `load` is waiting for all the remaining `stores` to commit because of an unresolved hazard. We perform an additional experiment with all store addresses replaced with non-aliased addresses except for one memory address to explore this further. This experiment shows that the peak disappears if there is only a single 4 kB, and 1 MB aliased address in the store buffer.

Lastly, we run the same experiments on a shuffled set of virtual addresses to assure that the contiguous virtual addresses may not affect the observed leakage. Our experiment with the shuffled virtual addresses exactly matches the same step-wise behavior suggesting that the upper bits in virtual addresses do not affect the leakage behavior. The leakage is solely due to the aliasing on physical address bits.

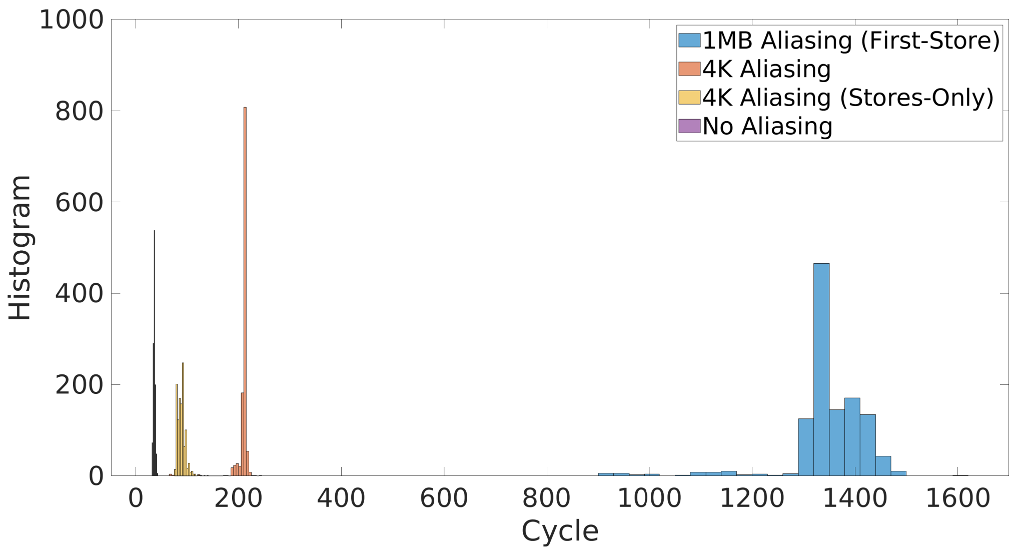


Figure 2.17: Histogram of the measurement for the speculative load with various store addresses. `Load` will be fast, 30 cycles, without any dependency. If there exists 4k aliasing only between the `stores`, the average is 100. The average is 200 when there is 4k aliasing of `load` and `stores`. The 1 MB aliasing has a distinctive high latency.

Comparison of address aliasing scenarios. We further test other address combinations to compare additional address aliasing scenarios using Algorithm 1. As shown by Figure 2.17, when `stores` and the `load` access different cache sets without aliasing, the `load` is executed in 30 cycles, which is the typical timing for an L1 data cache load including the `rdtscp` overhead. When the `stores` have different memory addresses with the same page offset, but the `load` has a different offset, the `load` takes 100 cycles to execute. Even memory addresses in the store buffer having 4k aliasing conditions unrelated to the speculative load create a memory bottleneck for the `load`.

In the next scenario, 4k aliasing between the `load` and all `stores`, the average load time is about 200 cycles. While the aforementioned 4k aliasing scenarios may leak cross-domain information about memory accesses (Section 2.4.4), the most unexpected behavior is the 1 MB aliasing, which takes more than 1200 cycles for the highest point in the peak. For simplicity, we refer to the 1 MB aliased address as *aliased address*, in the rest of this section.

The curious case of memory disambiguation. The CPU uses an additional speculative engine, called the *memory disambiguator* [92, 213], to predict false memory dependencies and reduce the chance of their occurrences. The main idea is to predict if a `load` is independent of preceding `stores` and proceed with the execution of the `load` by ignoring the store buffer. The predictor uses a hash table indexed with the address of the `load`, and each entry of the hash table has a saturating counter. If the pre-commitment dependency resolution does not detect false dependencies, the predictor increments the counter; otherwise, it resets the counter to zero. After multiple successful executions of the same `load` instruction, the predictor assumes that the `load` is safe to execute. Whenever the counter becomes zero, the predictor blocks nad check the next iteration of the `load` against the store buffer entries. Mispredictions result in performance overhead due to pipeline flushes. A *watchdog* monitors the prediction's success rate and temporarily disables the disambiguator to avoid repeated mispredictions.

The memory disambiguator's predictor should go into a stable state after the first few iterations since the memory `load` is always independent of any aliased store. Hence the saturating counter for the target speculative load address passes the threshold, and it never resets due to a false prediction. As a result, the memory disambiguator should always fetch the data into the cache without access to the store buffer. However, since the memory disambiguation performs speculation, the dependency resolution at some point verifies the prediction. The watchdog only disables the memory disambiguator when the misprediction rate is high, but in this case, we should have a high prediction rate. Accordingly, the observed leakage occurs after the disambiguation. During the last stages of dependency resolution, the memory disambiguator only performs prediction on the 4k aliasing at the initial loosenet check. Consequently, It cannot protect the pipeline from 1 MB aliasing appears at a later stage.

Hyperthreading effect. Motivated by MemJam [244], we empirically test to see if one can use the 1 MB aliasing as a covert/side channel through logical CPUs. Our observation shows that when we run our experiments on two logical CPUs on the same physical core, it causes the number of steps in the peaks to be halved. This finding matches the description by Intel [163], where it states that the core splits the store buffer between the logical CPUs. As a result, the 1 MB aliasing effect is not visible and exploitable across logical cores. However, we have seen that this observation is not valid for 4K aliasing, although in some Intel patents, they suggest that loosenet checks should mask out the `stores` on the opposite CPU thread.

2.4.3 Boosting Rowhammer and Cache Attacks with Spoiler

Knowledge of the physical address enables adversaries to weaken OS protections [199] and improve microarchitectural side-channel attacks [223]. Microarchitectural side-channel attacks such as rowhammer and cache attacks like the Prime+Probe rely on the virtual-to-physical address mapping [182, 273]. Without this knowledge, cache attacks such as

Prime+Probe on the Last-Level Cache (LLC) are challenging due to the runtime mapping of virtual addresses to cache sets and slices. Therefore, knowledge about the physical page mappings enables more attack opportunities using the Prime+Probe technique. Rowhammer [202] attacks require efficient access to rows within the same bank to induce fast row conflicts. An adversary needs to reverse engineer layers of abstraction from the virtual address space to DRAM cells to achieve this. The availability of physical address information facilitates this reverse engineering process.

Yet, the operating system should only allow root privileges to know about the physical address space. Previous attacks assume special privileges granted through weak software configurations to overcome some of these challenges [182, 223, 354]. For instance, the `procfs` filesystem exposes physical addresses [223], and *Huge pages* allocate contiguous physical memory [182, 226]. On mobile platforms, *Drammer* [354] exploits the Android *ION memory allocator* to access contiguous memory. *GLitch* [110] detects contiguous physical pages by exploiting row buffer conflicts.

Orthogonal to row buffer conflicts, the SPOILER leakage speeds up this reverse engineering of virtual-to-physical address mappings by a factor of 256 (due to 8-bit of leakage). This leakage translates to improving the Prime+Probe attack by a 4096-factor speed up of the eviction set search, even from sandboxed environments like JavaScript [187]. SPOILER is incredibly helpful for attacks in sandboxed low-privilege environments such as JavaScript, where previous methods require a time-consuming brute-forcing of the memory addresses [127, 265, 304]. Note that attacks are more limited in sandboxed environments since adversaries have limited access to the address space, and some instructions are also inaccessible [127]. However, SPOILER only relies on simple operations, `load` and `store`, to recover crucial physical address information, which enables rowhammer and cache attacks by leaking information about physical pages without assuming any weak configuration or special privileges. We now discuss that SPOILER can boost both single- and double-sided rowhammer attacks by its additional 8-bit physical address information and resulting detection of contiguous memory. For a more detailed analysis of using SPOILER for cache attacks and rowhammer, refer to the SPOILER paper [187].

DRAM bank co-location and single-sided rowhammer. For rowhammer, the adversary needs to access DRAM rows adjacent to a victim row efficiently. In a single-sided rowhammer attack, the attacker only activates one row repeatedly to induce bit flips on one of the nearby rows. For this purpose, the attacker needs to make sure that multiple virtual pages co-locate on the same bank. The probability of co-locating on the same bank is low without knowing physical addresses and their mapping to memory banks.

DRAMA [273] reverse engineered the memory controller mapping. This reverse engineering requires elevated privileges to access physical addresses from the `pagemap` file. The authors have suggested using the prefetch side-channel attacks to gain physical address information instead [126]. SPOILER is an alternative way to obtain partial address information and is still feasible when the `prefetch` instruction is not available, e.g., in JavaScript. We use SPOILER to detect aliased virtual memory addresses where the 20 least significant bits of the physical addresses match. The memory controller uses these bits for mapping the physical addresses to the DRAM banks [273]. Even though the memory controller may use additional bits, most of the bits are known using SPOILER. An attacker can directly hammer such aliased addresses to perform a more efficient single-sided rowhammer attack with a significantly increased probability of hitting the same bank.

To verify if our aliased virtual addresses co-locate on the same bank, we use the row conflict side channel as proposed in [110]. We observe that whenever the number of physical address bits used by the memory controller to map data to physical memory is equal to or less than 20, we always hit the same bank. For every extra bit that the memory controller uses, we can divide the probability of hitting the same bank by two as there is one more bit of entropy. In general, we can formulate that our probability p to hit the same bank is $p = 1/2^n$, where n is the number of unknown physical address bits in the mapping. In summary, SPOILER drastically improves the efficiency of finding addresses mapping to the same bank without administrative privilege or reverse engineering the memory controller mapping.

Multi-sided rowhammer. For a double-sided rowhammer attack, we need to hammer rows adjacent to the victim row in the same bank. The attacker tries to access two different rows $n+1$ an $n-1$ to induce bit flips in the row n placed between them. While double-sided rowhammer attacks induce bit flips faster due to the extra charge on the nearby cells of the victim row n, they further require access to contiguous memory pages. Alternatively, attackers can detect adjacent memory pages in the allocated virtual address space since such memory rows are written to the banks sequentially. We will not locate neighboring rows without contiguous memory since the memory controller maps the randomly chosen physical pages to random DRAM banks.

An attacker can use SPOILER to detect contiguous memory using 1 MB aliasing peaks. To prove this, we compare the physical frame numbers to the SPOILER leakage for 10000 different virtual pages allocated using `malloc`. Figure 2.18 shows the relation between 1 MB aliasing peaks, and physical page frame numbers. When the distance between the peaks is random, the trend of frame numbers also changes randomly. After around 5000 pages, we observe that the frame numbers increase sequentially. The number of pages between the peaks remains constant at 256, where this distance comes from the 8 bits of physical address leakage due to 1 MB aliasing.

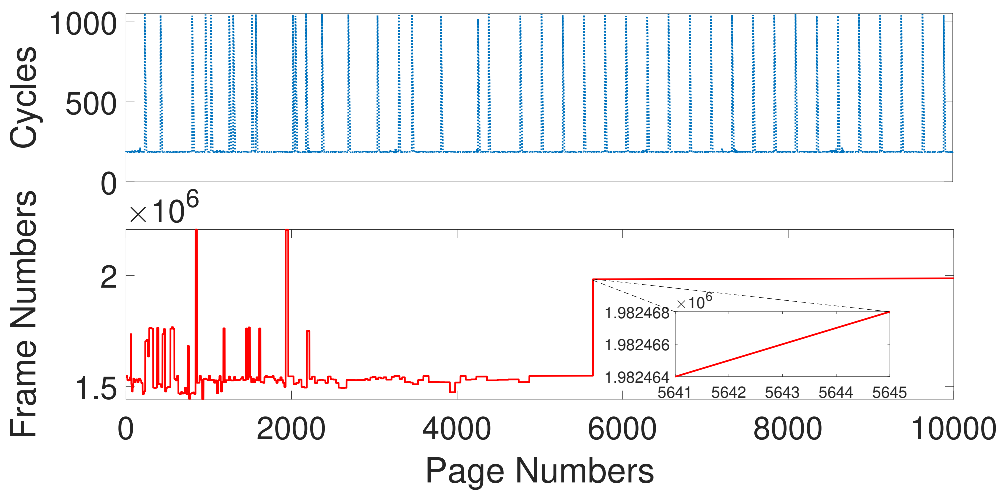


Figure 2.18: Relation between leakage peaks and the physical page numbers. The dotted plot shows the leakage peaks from the SPOILER attack. The continuous plot shows the physical frame numbers' decimal values from the `pagemap` file. Once the peaks (the dotted plot) become regular, the solid plot is linearly increasing, which shows contiguous memory allocation.

2.4.4 Tracking Speculative Loads with Spoiler

Single-threaded attacks allow stealing information from other security contexts running before/after the attacker code on the same thread [68, 245]. Example scenarios are context switches between different users' processes, between a user process and a kernel thread, and Intel Software Guard eXtensions (SGX) secure enclaves [245, 352]. In such attacks, adversaries bring the microarchitecture to a particular state and wait for the context switching to the victim thread. Next, they observe the microarchitectural state after the victim's execution and context switching back to the attacker's thread.

We propose an attack where the adversary **(1)** fills the store buffer with arbitrary addresses, **(2)** issues the victim context switch and lets the victim perform a secret-dependent memory access, and **(3)** measures the execution time of the victim. Any correlation between the victim's timing and the load address can leak secrets [385]. Due to the nature of the SPOILER, the victim should access a security-critical memory address. At the same time, there are aliased addresses in the store buffer, *i.e.*, if the `stores` are committed before the victim's speculative load, there will be no dependency resolution hazard.

To investigate the viability of SPOILER attack, we first analyze the depth of the operations that we can execute between the `stores` and the `load` In this experiment, we repeat several instructions between `stores` and the `load` that are free from memory operations. Figure 2.19 shows the number of stall steps due to the dependency hazard with the added instructions. Although `nop` is not supposed to take any cycle, adding 4000 `nop` will diffuse the timing latency. Then, we test `add` and `leal`, which use the Arithmetic Logic Unit (ALU) and the Address Generation Unit (AGU), respectively. Figure 2.19 shows that only 1000 `adds` can be executed between the `stores` and `load` before the SPOILER effect is lost. Since each `add` typically takes about 1 cycle to execute, this roughly gives a 1000 cycle depth for SPOILER. Considering the observed depth, we discuss potential attacks that can track the speculative load in the following two scenarios.

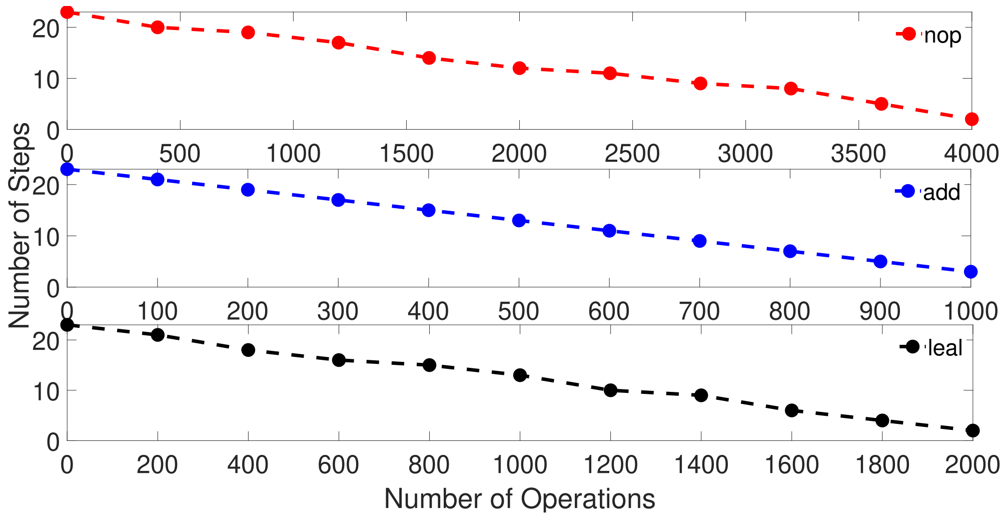


Figure 2.19: The depth of SPOILER leakage with respect to different instructions and execution units.

Context switching. In this case, we are interested in tracking memory access in the privileged kernel environment after a context switch. First, we fill the store buffer with addresses with the same page offset and then execute a system call. We expect to observe a delayed performance during the system call execution if a secret `load` address has aliased with the `stores`. We utilize SPOILER to iterate over various virtual pages; thus, some of the pages have more noticeable latency due to the 1 MB aliasing.

We analyze multiple `syscalls` with various execution times. For instance, Figure 2.20 shows the execution time for `mincore`. In the first experiment (red/1 MB Conflict), we fill the store buffer with addresses that have aliasing with a memory `load` operation in the kernel code space. The 1 MB aliasing delay with 7 steps suggests that we can track the address of a kernel memory `load` by the knowledge of our arbitrary filled store addresses. The blue (No Conflict) line shows the timing when there is no aliasing between the target memory `load` and the attackers `store`. Surprisingly, only by filling the store buffer, the system call executes much slower: the normal execution time for `mincore` should be around 250 cycles (cyan/No Store). This proof of concept shows that SPOILER can leak information from more privileged contexts; however, this is limited only to `loads` that appear at the beginning of the next context.

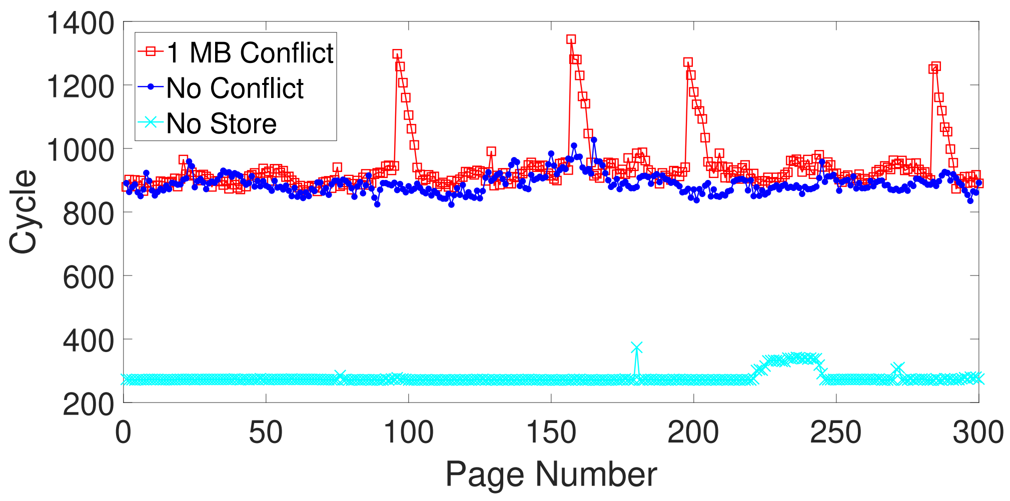


Figure 2.20: Execution time of `mincore` system call. When a kernel `load` address has aliasing with the attacker's `stores` (red/1MB Conflict), the step-wise delay will appear. These timings are measured with Kernel Page Table Isolation disabled.

Spoiler on SGX (negative result). In this experiment, we try to combine SPOILER with the *CacheZoom* [245] approach to creating a novel single-threaded side-channel attack against SGX enclaves with a high temporal and spatial resolution (4-byte) [244]. We use *SGX-STEP* [351] to precisely interrupt every single instruction. *Nemesis* [352] shows that the time to execute a context switch when an interrupt occurs depends on the execution time of the currently running instruction. On our test platform, *Core i7-8650U*, each context switch on an enclave takes about 12000 cycles to execute. If we fill the store buffer with memory addresses that match the page offset of a `load` inside the enclave in the interrupt handler, the context switch timing is increased to about 13500 cycles. While we cannot observe any correlation between the matched 4 kB, or 1 MB aliased addresses; we see unexpected periodic downward peaks with similar step-wise behavior as SPOILER (Figure 2.21).

We later reproduce a similar behavior by running SPOILER before an `ioctl` routine that flushes the TLB on each call. Intel SGX also performs an implicit TLB flush during each context switch. Thus, we can infer that the downward peaks occur due to the TLB flush, especially since these peaks' addresses do not have any address correlation with the address of the `load`. This behavior suggests that the TLB flush operation itself is affected by SPOILER and virtually eliminates the opportunity to observe any potential correlation due to the speculative load. As a result, we can not use SPOILER to track memory accesses inside an enclave.

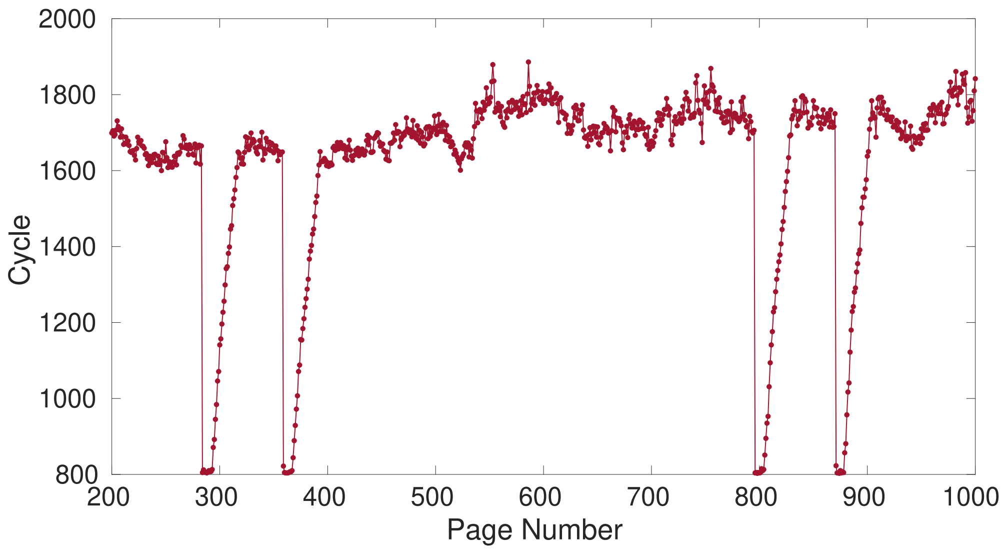


Figure 2.21: The effect of SPOILER on TLB flush. The execution cycle always increases for 4 kB aliased addresses, except for some of the virtual pages inside the store buffer where we observe step-wise hills.

2.5 Summary

First, this chapter contributes, with MEMJAM, a new side-channel attack based on false dependencies. For the first time, we discovered new aspects of this attack vector and its capabilities. MEMJAM uses false read-after-write dependencies to slow down accesses of the victim to a particular 4-byte memory block *within* a cache line. The state-of-the-art timing side-channel analysis techniques can exploit the resulting latency of otherwise constant-time implementations. We will show how to extract secrets from modern cryptographic implementations by applying the attack to recent implementations of 3-DES, AES, and SM4, as found in Intel IPP (§6.1). According to the available resources, the leakage source for the MEMJAM attack is present in all Intel CPU families released in the last ten years [107, 163], including newest generation CPUs. Our results also show that MEMJAM is a viable intra-cache-line attack applicable to SGX enclaves. Before MEMJAM, it might have seemed reasonable to design SGX enclaves under the paradigm that constant cache line accesses result in leakage-free code. However, the increased intra-cache-line granularity of MemJam shows that only code with real constant-time properties, *i.e.*, constant execution flow, and constant memory accesses, can be expected to have no remaining leakage on modern microarchitectures.

Next, we introduced Spoiler, a novel approach for gaining physical address information by exploiting a new information leakage due to speculative execution. To exploit the leakage, we used the speculative load behavior after jamming the store buffer. Spoiler can be executed from user space and requires no special privileges. We exploited the leakage to reveal information on the eight least significant physical page number bits, critical for many microarchitectural attacks such as rowhammer and cache attacks. We analyzed the causes of the discovered leakage in detail and showed how to exploit it to extract physical address information.

Gaining partial knowledge of the physical address will make such attacks feasible in browsers even though JavaScript-enabled attacks are challenging to realize in practice due to the limited nature of the JavaScript environment. We showed the impact of Spoiler to perform a rowhammer attack in a native user-level environment. Broadly put, the leakage described in this paper will enable attackers to perform existing attacks more efficiently or devise new attacks using the novel knowledge. For example, another area of work that may benefit from Spoiler or MemJam, and in general, are transient execution attacks like Meltdown [224]. We discuss this potential direction further in Section 3.1.

The source code for Spoiler is available on GitHub[1].

[1]`https://github.com/UzL-ITS/Spoiler`

Chapter 3

Microarchitectural Data Leakage via Automated Synthesis

We have contributed to a new class of transient execution attacks, called microarchitectural data sampling (MDS). Based on the Meltdown attack technique, MDS enables adversaries to leak secrets across security domains by collecting data from shared resources such as data cache, fill buffers, and store buffers within the CPU core. These CPU resources may temporarily hold data that belongs to other processes and privileged contexts, which the CPU could falsely forward to the adversary's memory accesses. Leaking data from these resources results in a variety of real-world attacks and security implications. Section 3.1 provides an overview of meltdown attacks, and in general, attacks that target transient execution.

Fuzzing is well known for finding vulnerabilities across trust boundaries [59, 64, 114, 190, 193, 210, 231, 235]. These approaches can usually rely on a well-defined interface, e.g., system calls. Previous work also investigated the use of fuzzing for finding Spectre gadgets [263]. With SpecFuzz, Oleksenko et al. [263] apply fuzzing techniques to find Spectre-PHT (also known as Spectre Variant 1) gadgets in existing code. However, they do not try to find new attack variants.

In Section 3.2, we perform an in-depth analysis of these Meltdown-style attacks based on a fuzzing-based approach. we introduce an analysis tool, named TRANSYNTHER, which mutates the basic block of existing Meltdown variants to generate and evaluate new Meltdown subvariants. We apply TRANSYNTHER to analyze modern CPUs and better understand the root cause of these attacks. As a result, we find new MDS variants that only target specific memory operations, e.g., fast string copies. Based on our findings, in Section 3.3, we propose a new attack technique, named MEDUSA, which can leak data from implicit write-combining (WC) memory operations. In Section 3.4.2, we discuss potential techniques to improve TRANSYNTHER and generalize the root cause for Meltdown attacks based on what we have learned in this chapter.

3.1 Transient-execution Attacks

As we have discussed in Section 2.1, modern CPUs employ out-of-order and speculative execution to increase performance. With out-of-order execution, CPUs can execute instructions further in the instruction stream as long as their dependencies are satisfied. These optimizations reduce the times a CPU has to stall due to long-running instructions significantly. Similarly, speculative execution enables a CPU to guess a conditional branch's outcome to continue executing the most likely path. If an instruction that was executed out of order or speculatively was based on an incorrect prediction, it is simply not committed to the architectural state. However, the instruction might have had a side effect on the microarchitectural state, such as the cache. In this case, such an instruction is called a transient instruction [62, 207, 224]. *Meltdown* [224] and *Spectre* [207] discover that such optimization techniques cause a new class of vulnerabilities, called transient-execution attacks. Transient-execution attacks exploit such transient instructions to leak data [62]. These attacks have since changed the perspective by introducing data leakage from the CPU rather than spying on access patterns, as we discussed earlier in Chapter 2.

3.1.1 Spectre & Meltdown

Spectre. Speculative engines predict an operation's outcome before its completion, and they enable the execution of the following dependent instructions ahead of time. As a result, the pipeline can maximize instruction-level parallelism and resource usage. In rare cases where the prediction is wrong, the pipeline needs to be flushed, resulting in performance penalties. However, this approach suffers from security weaknesses, in which an adversary can fool the predictor and introduce arbitrary mispredictions. These mispredictions execute illegal instructions that leave microarchitectural footprints in the cache. Adversaries can collect these footprints through the cache side channel to steal secrets e.g., using Flush+Reload [384].

Various variants of Spectre attacks frequently trick different types of branch predictors into executing control paths that are illegal architecturally [68, 148, 203, 207, 211, 227]. If the CPU cannot resolve a conditional branch's condition, it speculates where the execution continues. If this speculation was wrong, the CPU transiently executes instruction streams that should not architecturally happen. In such a case, the CPU might access the application data that should not be accessed (e.g., out-of-bounds values). Attackers can encode the accessed data into the microarchitectural state if a suitable Spectre gadget exists in the transiently executed instruction stream. Similarly to Meltdown (explained shortly), a covert channel can bring the microarchitectural state to the architectural form, leaking the secret [68, 148, 203, 207, 211, 227]. *Spectre* attacks on the branch prediction unit [207, 227] imply that one can use side channels such as caches as a primitive for more advanced attacks on speculative engines.

Meltdown. Meltdown attacks [61, 224, 297, 300, 339, 348, 356] exploit the heavily optimized out-of-order load operations in which faulting memory loads still proceed with stale or illegal data. In Meltdown attacks, the attacker leverages out-of-order execution following a faulting load [224]. In contrast to Spectre, in which instruction sequences transiently access secrets in the same security domain, Meltdown allows accessing secrets across a security boundary. Transient execution of instructions after a fault, as exploited by Lipp et al. [224] and Bulck et al. [348], can leak memory content of protected environments. Similarly, transient behavior due to the lazy store/restore of the FPU and SIMD registers can leak register contents from other contexts [322]. Using a covert channel, such as Flush+Reload, the attacker can then bring the microarchitectural state to the architectural form, ultimately leaking the secret.

3.1.2 Microarchitectural Data Sampling

Canella et al. [62] have systematically analyzed new variants of both Meltdown and Spectre and propose a generic taxonomy to classify transient-execution attacks. They based their classification on the cause of the transient-instruction sequence and the exploited microarchitectural buffer. While this classification captures the cause and targets of known variants in a structured way, it does not inform how specific attacks are triggered. For most Meltdown attacks, there are multiple ways to trigger the leakage, e.g., some variants seem to require TSX to suppress exceptions [366] while others can also leverage signal handlers or misspeculation [61, 224, 300]. Meltdown-type attacks exploit complex situations in the microarchitecture, which require so-called microcode assists. Microcode assists are software routines in the CPU to handle cases that hardware logic cannot directly address, e.g., specific faults, or updating bits in page-table entries.

We have contributed to a class of Meltdown-style attacks collectively referred to as microarchitectural data sampling (MDS) [61, 300]. MDS also highlights the negative impact of Intel hyperthreading on the security of modern CPUs [241, 297, 300, 322, 356, 366]. However, the performance gained by SMT proved indispensable outweighing any data leakage across logical CPUs. Intel has promised fixes for such critical issues on all future CPUs. While hardware patches have been deployed for data spilling attacks such as MDS, side channels across logical CPUs remain a valid concern without foreseeable hardware mitigations. These attacks can thrive in ever-expanding use cases of core multitenancy such as the Platform as a Service (PaaS) cloud [345].

Address aliasing for data sampling. We mentioned that address aliasings introduce new microarchitectural side channels (§2). We also have demonstrated that SPOILER can improve other microarchitectural attacks like cache attack and rowhammer. Additionally, we have seen in *Spectre-STL* [148] and our work on the *Fallout* attack [61] that these logics for handling memory addresses relate to some of the transient execution attacks. Spectre-STL relies on memory disambiguator to transiently bypass memory stores followed by a memory load even if the `load` has an actual dependency on the `store`. As a result, an attacker can leak the previous (stale) value of the `load` even if the prior `store` meant to overwrite this leaked value. Unlike Spectre-STL, Fallout exploits faults and microcode assists. In Fallout, we show that when a memory load causes a microcode assist or an architectural exception and the target store operation have 4 k aliasing with this faulty/assisted memory load, similar to MEMJAM, it may leak data from the store buffer. Although this shows that address aliasings have a direct impact on Meltdown-style attacks and particularly MDS, we have later verified that 4k aliasing is not always needed for leaking data from store operations [247]. Before discovering the Fallout, we have conducted the following experiment to test the possibility of leaking data from the store buffer without an architectural fault or microcode assist and by just exploiting address aliasing conditions described in MEMJAM and SPOILER. We

1. introduced either 4K aliasing (MEMJAM) or 1 MB aliasing (SPOILER) between several store operations with a known value and a load operation,
2. measured the time for the memory load operations and checked if the timings for each condition matches with Figure 2.17,
3. similar to Meltdown, encoded the byte value of the memory load operation into a cache line and scanned all the possible 256 possible caches lines using the Flush+Reload technique.

Although intuitively, the `load` should consume falsely-forwarded data from the aliased `store`, our experiments demonstrated negative results. We observed that the `load` always encodes the correct value into the cache even when we observe the timing behavior corresponding 4 k aliasing and 1 MB aliasing. Therefore, leaking data from the aliased `store` seemed not feasible without a faulty/assisted load. On the other hand, when we change the `load` to experience a fault or assist, we observe the store buffer data leakage described in Fallout. One explanation for this behavior is that the invalid forwarding due to address aliasing, by default, may not provide a big-enough speculative window for the following operations and the secondary memory load to encode the wrong value into the cache covert channel, but the condition for such data leakage is suitable during a fault or an assist.

Although our experiments at the time suggested that store buffer data leakage without a fault or microcode assist was not possible on the tested Kaby Lake and Skylake architectures, a recent security advisory refers to the *fast forward store predictor* [156]. We are not sure about the detail of this recently discovered vulnerability on Intel CPUs, as there is no public description of the requirement for this new finding by Intel engineers. However, this advisory indicates that leaking data from store operations solely abusing predictors is feasible on some microarchitectures. Based on this report, we speculate that either the experiment we have designed was insufficient to demonstrate predictor-based store leakage, or this vulnerability is only applicable on more recent CPUs, *i.e.*, microarchitectures and microcode versions that we have not tested. In conclusion, there are apparent correlations between address aliasings and transient execution attacks. We suggest the community to investigate such correlations further in open-source hardware.

Load value injection. Following MDS, we also have contributed another set of attacks to the transient execution attack taxonomy. Load value injection (LVI) [349] demonstrates an inverse-Meltdown attack, in which the adversary exploits microarchitectural buffers to inject invalid data and control flow within the pipeline. When adversaries combine this filling up the microarchitectural buffers with introducing intentional faults or microarchitectural assists to a target memory load, they can construct gadget-based data exfiltration techniques Similarly to Spectre.

In theory, LVI attacks impact several threat models, including cross-process, user-to-kernel attacks, and attacks against TEEs like Intel SGX. However, in practice, due to several timing constraints during an attack, they have only impacted workloads like Intel SGX. SGX threat model considers the operating system as malicious. We will look into the threat model of adversarial operating systems against SGX in Chapter 4.

3.2 Automatically Exploring Meltdown Attacks

This section proposes a systematic approach for evaluating data leakage caused by the combination of microcode assists caused by a `load` with dependent operations. For this purpose, we build TRANSYNTHER[1], a tool to automatically generate and test the combination of known building blocks of Meltdown attacks with new triggers of

[1]TRANSYNTHER tool and MEDUSA attack codes are available as open-source implementation on GitHub: https://github.com/vernamlab/Medusa

microcode assists. Furthermore, we use fuzzing-type techniques to mutate, evolve, and combine building blocks. TRANSYNTHER can automatically evaluate whether the newly synthesized code variants are a variant of a Meltdown attack by trying to leak known values.

3.2.1 Introducing Transynther

In this section, we introduce TRANSYNTHER, an automated approach for exploring Meltdown-type attacks. TRANSYNTHER uses fuzzing-based techniques to explore Meltdown-type attacks systematically. The aim is to identify new variants of existing attacks which are, e.g., faster, less complicated, or are still exploitable mitigated, as well as entirely novel Meltdown-type attacks.

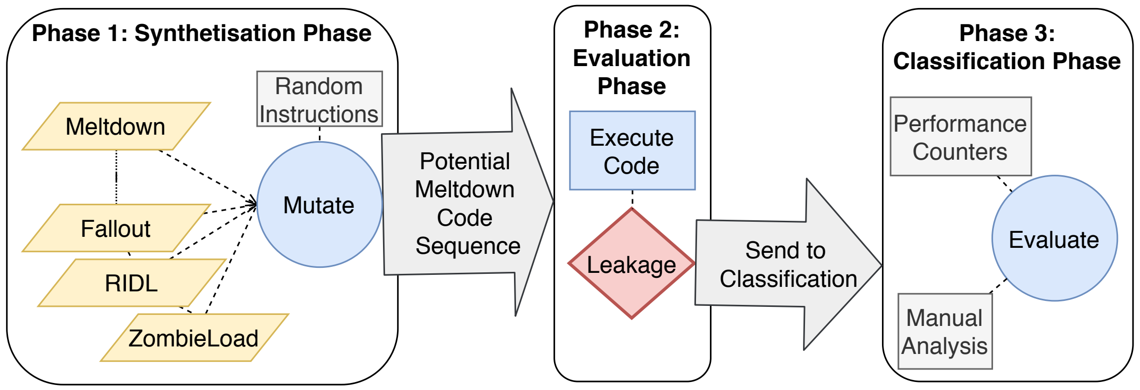


Figure 3.1: TRANSYNTHER phases: After mutating a new code sequence for a meltdown-style attack, the code is evaluated. If there is a leakage detected, the sample is analyzed further during the classification phase.

TRANSYNTHER works in three phases, as outlined in Figure 3.1. In the first phase, the *synthetisation phase*, TRANSYNTHER uses building blocks of existing attacks to mutate and combine them to potential new attacks. In the second phase, the *evaluation phase*, TRANSYNTHER executes the code from the synthetisation phase and evaluates whether the code leads to data leakage. Finally, if the evaluation phase was successful, the *classification phase* tries to classify the leakage source using performance counters automatically.

3.2.2 Synthetisation Phase

The first phase is the synthetisation phase. In this phase, TRANSYNTHER generates a code snippet, which is a potential Meltdown-type attack. For this, TRANSYNTHER relies on building block from existing Meltdown-type attacks, including Meltdown [224], ZombieLoad [300], RIDL [356], Foreshadow [348], Fallout [61], Meltdown-PK [62], Meltdown-AVX [157], and Meltdown-RW [203].

The common pattern for all these attacks is as follows

① Prepare the microarchitectural state (e.g., flushing, accessing, or storing data).

② Load operation causing a fault (as Schwarz et al. [300], we consider microcode assists as microarchitectural faults).

③ Dependent operation consuming the loaded data and encoding it in a microarchitectural element.

As the encoding in ③ does not affect the leakage of a Meltdown-type attack [224, 339], we always choose to encode the loaded value in the cache. This choice allows us to recover the encoded values using a Flush+Reload covert channel quickly. This approach is used for the majority of Meltdown-type attacks [61, 62, 203, 224, 300, 322, 348, 356, 366]. Initially, TRANSYNTHER sets up and uses two pools in ②. One pool contains possible *load operations* and one contains possible *load targets*:

Load operations. Memory Loads are operations that load data from memory addresses into registers. The simplest case of a load operation is a `mov` from a memory address to a general-purpose register. TRANSYNTHER relies on such `mov`s for all possible sizes, from 8 to 64 bit. Additionally, possible load operations are also aligned and unaligned AVX loads (`{v}movaps`/`{v}movups`) with a size of 128 and 256 bit.

Load targets. Load targets are virtual addresses with a systematic pattern of page-table-entry bits, as discussed by Canella et al. [62]. As a starting point for this pool, we rely on load targets with specific page-table bits, which Meltdown-type attacks already used. This includes the user-accessible bit [224, 300], accessed bit [61, 300], present bit [61, 348, 356, 366], writable bit [203], and protection key [62]. For a systematic approach, we furthermore add load targets with page-table bits that have not been used in successful Meltdown-type attacks so far, including the dirty bit, write-through bit, uncachable bit, size bit, and non-executable bit. For all addresses, we define the content of the corresponding physical page to evaluate the leakage. Finally, we also add addresses that do not have a valid mapping to physical pages, such as non-canonical addresses (addresses where the bits 48 to 63 are different to bit 47, e.g., `0x1234567812345000`) and `NULL`.

Furthermore, TRANSYNTHER creates a victim that leaks specific data by repeatedly loading and storing it to different virtual addresses and memory types. The victim can either be a separate application running on the sibling hyperthread or running time-sliced on the same hyperthread, e.g., using multithreading.

During synthesis, TRANSYNTHER randomly chooses, mutates, and combines building blocks for ① and ②. TRANSYNTHER randomly chooses an operation (load, store, or flush) and an address from the load-target pool to prepare the microarchitecture (①). Then, it mutates the address by adding a random offset between 0 B and 4 kB. This mutation ensures that the address still maps to the same page in most cases because of the page offset to a different cache line. Note that there is the case that a multi-byte load might lead to a split-page load if the offset is too large. The split-page load is the intended behavior, as split-page loads are also corner cases that might lead to leakage. For ②, TRANSYNTHER randomly chooses a load operation and a load target. Again, we randomly choose the added offset between 0 B and 4 kB.

Additionally, TRANSYNTHER randomly inserts independent operations between the preparation of the microarchitecture (①) and the faulting load (②). Such operations are, e.g., `nops` (no operations), ALU operations on unrelated registers, and additional architectural faults. These instructions add a certain amount of timing differences and increase the chance of triggering a race condition in the hardware. These operations may increase the leakage rate for existing attacks, as shown in the published proof-of-concept implementations for other transient-execution attacks [62, 224, 300].

Finally, TRANSYNTHER adds another load operation, consuming the faulting load's value in ② and encoding it into the cache. In line with previous work [62, 207, 224, 300, 322, 356], this operation simply accesses the n^{th} page in a 256-page array, where n is the byte value provided by the faulting load in ②. Again, TRANSYNTHER can randomly insert independent operations between this step and the faulting load to vary the timing between ② and ③.

3.2.3 Evaluation phase

In the second phase, the evaluation phase, TRANSYNTHER evaluates whether the synthesized code snippets from the synthetisation phase lead to data leakage. TRANSYNTHER uses an evaluation framework consisting of a preparation part which fills microarchitectural buffers, the synthesized code snippet augmented with exception suppression, and a Flush+Reload loop to recover the values encoded in ③. The evaluation framework's code runs in an endless loop for a user-specified amount of time, e.g., 2 seconds. Then it compares the values recovered using Flush+Reload to the known values from the preparation part. For every evaluated snippet, TRANSYNTHER logs the number of correct and wrong leaked values. Snippets for which correct leakage is detected are *candidate snippets* used in the classification phase. Snippets that do not leak correct values are discarded and not further analyzed. In contrast to traditional application fuzzing, there is no feedback in our approach, enabling TRANSYNTHER to improve a snippet. The only feedback that the

CPU provides is whether the snippet leaks data or not. Moreover, as we try to discover vulnerabilities, TRANSYNTHER cannot use a CPU emulator [232].

3.2.4 Classification Phase

In the final phase, TRANSYNTHER analyzes the leakage source using *microarchitectural buffer grooming*, and *performance counters*.

Microarchitectural buffer grooming. The main idea of microarchitectural buffer grooming is to bring these buffers into a known state. To achieve this, we fill every microarchitectural buffer with known data that is unique for each buffer. Hence, if we observe any leakage, we can infer the leakage source from the values. In the simplest case, each buffer contains a repeated, single printable character. For example, by storing several 'S'-characters, we "fill" the store buffer with this character. If we then leak multiple 'S'-characters, we can consider the store buffer as a potential leakage source. By having a unique character per buffer, buffer grooming provides an elementary form of data taint tracking [30]. In the case of data leakage, TRANSYNTHER at least knows the data origin.

We only consider on-core data buffers for the buffer grooming, *i.e.*, the L1 data cache, store buffer, line-fill buffers, load buffer, load ports, and WC buffers. While buffer grooming is straightforward for some buffers, e.g., the L1 cache, it is more difficult for other buffers, e.g., the line-fill buffer. Fortunately, Intel provides software sequences for mitigating some of the MDS attacks. These software sequences are designed to zero-out the data in all microarchitectural data buffers [157], *i.e.*, it sets the values in all buffers to zero.

Listing 1 shows the software sequence used to zero-out the buffers on Skylake and newer microarchitectures. In Lines 3 to 4, the load ports are zeroed out. Then, 12 cache lines are flushed (Line 6) to ensure that 12 of the subsequent writes in Line 13 have to go through the 12 line-fill-buffer entries [300]. Using `rep stosb` also ensures that the WC-buffer entries of the line-fill buffer are also used and thus zeroed-out. For the buffer grooming, we can rely on an adapted software sequence. Instead of writing zero to all buffers, we store a repeated, unique character to every buffer. This procedure is as simple as, e.g., letting `zero_ptr` point to a memory content not containing 0 but 'L'-characters to ensure that load port is filled with repeating 'L's. Moreover, we can replace the `rep stosb` with a normal `mov` in a loop to distinguish WC buffers from general line-fill buffers.

The obvious limitation is that TRANSYNTHER cannot track the actual flow of the data in hardware. For example, the CPU may have already written the store buffer's data to the L1 cache. Therefore the data may also leak from the L1 cache. Still, TRANSYNTHER

Listing 1 Software sequence to overwrite all microarchitectural buffers for Skylake and newer microarchitectures [157].

```
mov %[scratch], %rdi
lfence
orpd (%[zero_ptr]), %xmm0
orpd (%[zero_ptr]), %xmm0
xorl %eax, %eax
1: clflushopt 5376(%[scratch],%rax,8)
addl $8, %eax
cmpl $8*12, %eax
jb 1b
sfence
movl $6144, %ecx
xorl %eax, %eax
rep stosb
mfence
```

Counter	Description
MEM_LOAD_RETIRED.FB_HIT	Data loaded from a line-fill-buffer entry.
MEM_LOAD_RETIRED.L1_HIT	Data loaded from the L1 data cache.
MEM_LOAD_RETIRED.L2_HIT	Data loaded from the L2 data cache.
L1D_PEND_MISS.FB_FULL	Data is neither in L1 nor in fill buffer.
LD_BLOCKS.STORE_FORWARD	Store buffer blocks load.
LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	Load blocked by partial address match.
MEM_INST_RETIRED.SPLIT_LOADS	Data spans across two cache lines.

Table 3.1: The performance counters used in Transynther to identify the active microarchitectural elements.

assumes that the data leakage is from the store buffer. To reduce the number of false classifications, we additionally rely on hardware performance counters.

Performance counters. For additional information on the leakage source, we augment Transynther to record hardware performance counters while leaking values. Thus, in addition to the source of leaked values, we also observe the active microarchitectural elements.

Table 3.1 shows the performance counters we used. Some of these performance counters can identify leakage sources successfully [187, 300]. They cover multiple microarchitectural buffers, such as the line-fill buffer, L1 and L2 data cache, and the store buffer. Figure 3.2 shows the heatmap for the correlation between the number of leaked bytes and different performance counter events related to various variants of Meltdown attacks. The darker the color is, there is more relative correlation.

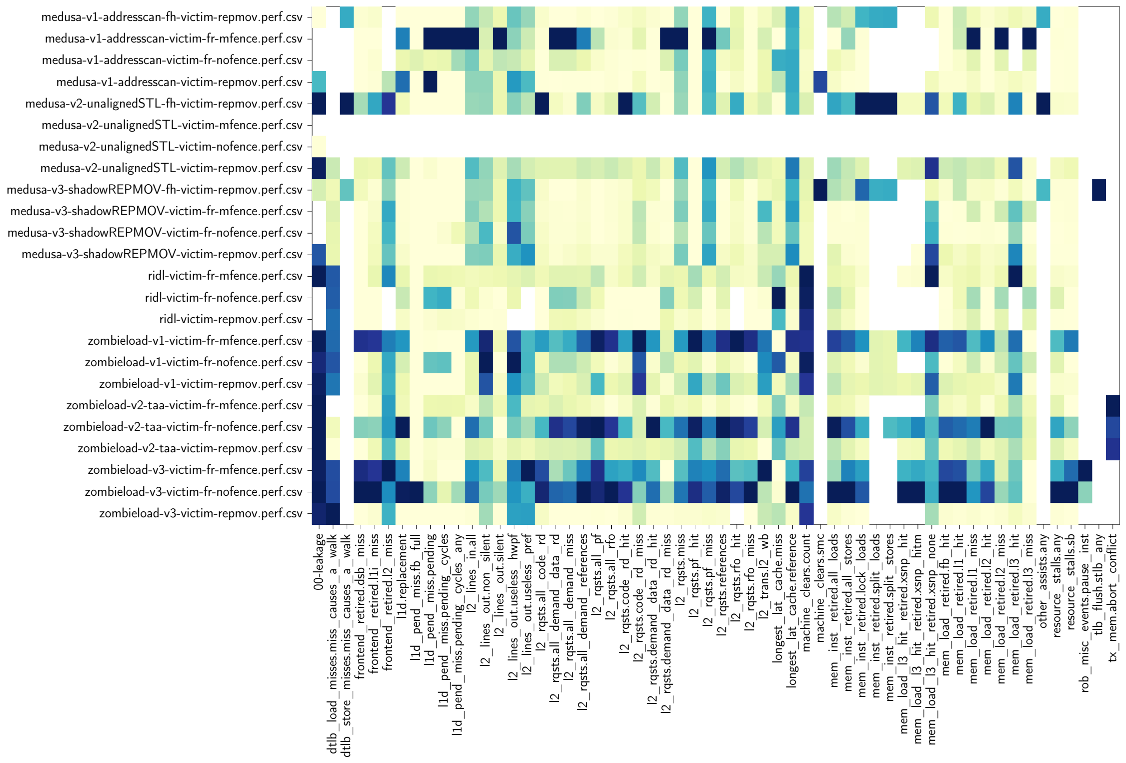


Figure 3.2: Heatmap of performance counters

Transynther correlates the performance-counter values with the number of leaked bytes using the Pearson correlation coefficient. A high positive correlation between the number of leaked bytes and the events for a microarchitectural element indicates that this element is involved in the leakage.

With microarchitectural buffer grooming and the correlation coefficient from the performance counters, Transynther can provide an educated guess of the leakage source.

3.2.5 Transynther Results

In our first set of experiments on Intel CPUs, we ran Transynther for about 46 500 test cases distributed on the three Intel Core i7-7700 (Kaby Lake), i7-8650U (Kaby Lake R), and i9-9900K (Coffee Lake) CPUs. We ran each test case for 2 s, totaling about 26 CPU hours. Transynther generated 5100 code snippets, which showed transient leakage. Based on the classification and subsequent manual analysis, we filtered the

CPU	**μarch**
Intel Core i7-7700	Kaby Lake
Intel Core i7-8650U	Kaby Lake R
Intel Core i9-9900K	Coffee Lake
AMD Ryzen 5 2500U	Ryzen 5

Table 3.2: Tested Environments.

generated code snippet to 100 interesting cases with a unique code and leakage pattern. We identified multiple classes of leaking code sequences.

We also ran some tests on an AMD Ryzen 5 2500U and show that while there is no data leakage on AMD, AMD is not by-design immune to Meltdown-type attacks. In our second experiment, we ran TRANSYNTHER for about 10 000 test cases on an AMD machine. Similarly, we ran each test case for 2 s, totaling about 5 CPU hours. We present our findings in Intel CPUs, followed by our conclusions regarding AMD CPUs.

Split cache access. TRANSYNTHER reproduced various variants of split cache access that lead to MLPDS. Split accesses refer to memory accesses that span over two cache lines and are handled differently from normal loads accessing a single cache line. In the generated proof of concepts, we can observe that when split access is suffering a faulty load, it directly leaks the data that is loaded by the sibling thread (①). Split access works for page faults (user-accessible and present) and microcode assists caused by setting the accessed bit. We only saw MLPDS leakage on Kaby Lake and Kaby Lake R but not on the Coffee Lake microarchitecture. Another observation is that MLPDS with split access works much faster when a page fault is caused by accessing a non-present page before the target faulty load. Unlike non-canonical addresses, Intel microarchitectures treat the zero address as non-present pages. In contrast, a page fault caused by accessing a non-user-accessible page does not increase the leakage rate. Vector move instructions can also trigger split accesses (②), which lead to the same leakage. In this case, a 16-, 32- or 64-byte vector move instruction has a higher chance of being unaligned.

Vector move. A faulting vector load instruction with correct alignment without crossing a cache line can leak data (③). Vector load instructions can enforce alignment e.g., `movaps` or be alignment-agnostic e.g., `movups`. A correct alignment means that either the load's address is aligned or the memory load is an alignment-agnostic instruction. Depending on which part of the vector is loaded, it can leak different bytes of the implicitly write-combined data. Prior faults may also affect which part of the data is leaked. We

Case	Preparation	Store	Load	Name
①	(access ⊘ , random instructions)	-	≺ + 🔒 / 👆 / ⊘	MLPDS
②	(access ⊘ , random instructions)	-	AVX ≺ + 🔒 / 👆 / ⊘	MLPDS
③	(access ⊘ , random instructions)	-	AVX + 🔒 / 👆 / <✖>	MEDUSA
④	(access ⊘ , random instructions)	-	AVX ⇉ + 🔒 / 👆 / ⊘ / <✖> / ✓	MEDUSA
⑤	-	store (to load)	🔒 / 👆 / <✖> / ✓	S2L
⑥	(`rep mov` + store, store + fence + load)	store (to load)	🔒 / 👆 / <✖> / ✓	-
⑦	-	store (4k aliasing) + 🔒 / 👆 / ⊘ / <✖> / ✓	🔒 / 👆	MSBDS
⑧	-	store (4k aliasing, to load) + 🔒 / 👆 / ⊘ / <✖> / ✓	AVX ⇉ + 🔒 / 👆 / ⊘ / <✖> / ✓	MSBDS, S2L
⑨	(Sibling on/off)	store (random address) + ⊘	🔒 / <✖>	MSBDS
⑩	(Sibling on/off + clflush (store address))	store (Cache Offset of Load) + ⊘	🔒 / <✖>	MSBDS
⑪	(Sibling on/off + `rep mov` (to Load))	store (to Load)	AVX ⇉ + 🔒 / 👆 / ⊘ / <✖> / ✓	MEDUSA, MLPDS
⑫	-	Store (Unaligned to Load)	🔒 / 👆 / <✖>	MEDUSA
⑬	(random instructions)	AVX Store (to Load)	<✖>	MEDUSA, MLPDS, MSBDS
⑭	-	random fill stores	<✖>	MSBDS

<✖> Non-canonical Address Fault ⊘ Non-present Page Fault 🔒 Supervisor Protection Fault
⇉ AVX Alignment Fault 👆 Access-bit Assist ≺ Split-Cache Access Assist ✓ Access without Fault/Assist

Table 3.3: Leakage variants discovered by TRANSYNTHER.

hypothesize that this is due to the different times it takes to handle the exception for the fast string copy operation. Faulting vector load operations also show fast leakage for a non-canonical address, whereas a simple non-canonical fault requires additional memory grooming to work. We did not observe leakage for a page fault in our setup of microarchitectural buffer grooming, in contrast, to split cache accesses. Note that while Intel refers to all these cases as MLPDS [157], we distinguish the specific case of leaking from implicit WC. On the other hand, a vector move with split access leaks any load operation on the sibling thread (②).

AVX alignment fault. When one uses an aligned version of a vector instruction, the provided address should be aligned with the memory request's size. Otherwise, the CPU throws a general-purpose exception. TRANSYNTHER created many variants of alignment-enforcing vector loads combined with unaligned addresses, leading to a general-protection exception. The results indicate that the alignment exception is prioritized in the pipeline as it does not depend on the address type (④). In contrast to ③, ④ also works with page faults and even valid addresses, not causing any faults. As we observed in other cases as well, grooming the pipeline by introducing early exceptions or adding random instructions of the pipeline may improve the leakage rate.

Store-to-leak. Transynther generated various variants of Store-to-leak [297]. Transynther showed that during a TSX transaction, Store-to-leak [62] works on all addresses except for non-present addresses (⑤). Accesses to non-accessed pages abort the TSX transaction, showing TAA [175].

Transynther also generated a case that when the CPU executes an unrelated `rep mov` instruction before the store, Store-to-leak does not forward the data anymore. We further noticed that adding a fence instruction between the store and load prevents Store-to-leak. For Fallout [61], it has no effect (⑥).

4K-Aliasing forwarding (Fallout). As shown in Fallout [61], store-to-load forwarding can forward the wrong data when the least-significant 12 bits of the store and load address match [244]. Transynther reproduced combinations of addresses that can forward when the store and load are a multiple of 4 kB apart (⑦). We verified that false forwarding on 4 kB aliasing only works with supervisor fault and access-bit assist. Transynther showed that the forwarding is agnostic to the store's address, *i.e.*, any store regardless of whether the target is a valid or invalid address is forwarded as far as it meets the 4 kB aliasing condition.

Store-to-load forwarding and AVX. In our experiments, both Fallout and Store-to-leak [61] also work with aligned AVX loads. However, when the load suffers a vector alignment general-protection exception, Store-to-leak and Fallout both ignore the address types for both stores and loads (⑧).

Store forwarding and faulting stores. Medusa discovered that faulting stores could be forwarded independently of address aliasing and matching. In ⑨, we perform a store to non-present addresses causing a page fault, e.g., address null. When the sibling thread is turned on and off, the store is forwarded to the faulting load without any aliasing. Interestingly, we can still choose which byte of the store to leak by indexing into different offsets. For instance, if the store's size is 8 B, we can choose which byte to leak by changing the last 3 bits of the faulty store or load. This variant of MSBDS only works with supervisor fault and non-canonical address exceptions. Also, if we perform multiple faulty stores, we leak more often from the later stores.

According to Intel, a sibling thread can leak stores from another thread when the other thread goes to sleep and back [157]. Potentially, this variant can create a basis for leaking stale data from a sibling store. Besides, this can potentially increase the attack surface for the exploitation of LVI attacks [349]. As in some LVI variants, attackers who control the store address may freely perform stores to faulty pages to inject data to the store buffer.

Store forwarding and cache Aliasing. Transynther also created code sequences that leak the store data based on aliasing of only the cache offset. This finding is in contrast to the current understanding that only full address matching or 4 kB aliasing forwards the data (⑩).

Store forwarding and stale load forwarding. As we mentioned in various cases, grooming the pipeline may affect which data will be forwarded/leaked first. For instance, Medusa generated many proof of concepts that shows we can leak different buffers and values with the vector alignment exception. We only mention one example here: one can turn Store-to-Leak into a case where both the store and a value from the sibling thread (MLPDS or Medusa) are leaked. In this case, we prepared the architecture with a `rep mov` instruction with the destination address being the faulty load address. When the sibling thread is switching on/off, we see that both the forwarded store and the values loaded by the sibling thread are leaked (⑪).

In this proof-of-concept, `rep mov` handled by a specific microcode assist [163] is causing the value from a sibling thread to be loaded instead of the expected Store forwarding, *i.e.*, the value stored previously. We investigated the effect of `rep mov` and found out that we can use it to create a new variant of leakage from the WC buffer (Section 3.3.2).

Unaligned store forwarding. We also found using Transynther that unaligned store forwardings can leak values from a sibling thread. This variant is a particular case of store-forward in which the store and load overlap partially, but it can not forward the actual data bytes on the store to the load operation. We investigate this case further and use it as a new attack variant for Medusa in Section 3.3.2 (⑫).

Non-canonical addresses. Non-canonical addresses are handled differently from regular memory addresses on Intel CPUs [335]. During an early stage of address decoding, the CPU converts a 64-bit address to a compacted form, as the actual supported address space is not 64-bit. If the address does not follow the canonical form [177], the CPU will throw a general-purpose exception during this conversion. We verified that no page table walk for non-canonical addresses and an early mechanism throw an exception matching the patent description.

Medusa observed various cases where the combination of non-canonical address faults will leak data with different behavior. For instance, store-to-leak on a no-canonical address may not always leak the value of the store. Instead, depending on how grooming affects the architecture, we see the store and loads from the sibling thread are leaked (⑬). Another interesting observation is that sometimes a non-canonical fault would always leak the last store, disregarding any aliasing form. In this case, we have filled the store buffer with various valid stores, and depending on what state the store buffer will be (a different set of random stores), there are cases where the CPU always forwards the last store to the load ($\textcircled{14}$).

Exception bypass (AMD). One of the requirements for Meltdown-type attacks is to bypass exceptions in an out-of-order fashion. The results from TRANSYNTHER suggest that the AMD Zen microarchitecture might potentially be vulnerable to Meltdown-type attacks. We found that various exceptions, such as division by zero, an aligned vector store general-purpose exception, and a faulting store to a supervisor address, do not stop the out-of-order execution. Instead, either the store to a valid (non-faulty) load after the fault was complete or the proper load operations leak the content with valid permission. In line with the AMD whitepaper [20], their CPU may bypass some of the exceptions speculatively. Hence, Meltdown's requirement is also present on AMD CPUs, the forwarding of data from faulting instructions. CPUs immune to Meltdown-type attacks have to ensure that operations depending on a faulting instruction cannot get the transient data, e.g., by stalling. While AMD provides this insurance for page faults, they do not guarantee this property for other faulting instructions, e.g., General Protection Memory Access (cf. AMD whitepaper [20], page 5). While we could not show data leakage that violates a security guarantee, e.g., leakage from the kernel, AMD is not by-design immune to Meltdown-type attacks.

Vector move alignment fault (AMD). We also observed that the AMD CPUs handle faulty vector alignment exceptions differently than other faulty loads. In particular, these exceptions do not block the data flow, and we observe that the pipeline will still speculatively consume the data despite the exception. We observe that the memory page's value or the recently written value to the memory page will be leaked using a Meltdown-style gadget. Again, this does not violate any architectural data flow, but it shows that computation over transient data that was not supposed to be available is feasible from a microarchitectural standpoint.

3.3 Medusa: Pre-filtering Data

In addition to reproducing known attacks and gaining a better understanding of the root cause, TRANSYNTHER also discovered new variations of MDS variants, which we refer to as MEDUSA. MEDUSA provides more in-depth insight into how Intel microarchitectures implement the memory subsystem. MEDUSA specifically targets data values transferred via the common data bus but are not normal data loads. In addition to AVX2 loads, MEDUSA has the unique property to observe the inner workings of implicit write combining (WC) used by the CPU, e.g., fast string operations like `rep mov`. For WC, the CPU uses a part of the line-fill buffer to combine multiple stores to the same cache line to increase the throughput. In contrast to ZombieLoad [300] and RIDL [356], which leak arbitrary data from the line-fill buffer, MEDUSA specifically targets data transfers caused by WC. WC memory semantics are explicitly used by memory marked as WC, and implicitly by the fast string operations, *i.e.*, `rep mov` and `rep sto`.

With MEDUSA, the leakage is extremely targeted and noise-free, as it only leaks specific memory operations. Thus, while the property to only leak data from WC sounds like a limitation, it is advantageous over previous data-sampling attacks. Data-sampling attacks such as ZombieLoad [300] or RIDL [356] required extensive post processing to find the targeted data within the leaked data, MEDUSA does not consider such large amounts of unrelated data in the first place. This attack primitive is incredibly essential, as ZombieLoad and RIDL, in practice, leaks too many unrelated data when applied to applications that perform lots of operations. For instance, in Section 6.2, our case study of attacking RSA, the computation of RSA, including loading the key from the disk and performing signing operations, consists of thousands of load operations. Attackers are not generally interested in leaking all of these load operations.

In this section, we further evaluate a novel ZombieLoad variant, which we discovered using TRANSYNTHER. First, we show that MEDUSA allows prefiltering leaked values. MEDUSA only leaks values used in implicit WC by exploiting the WC buffer's microarchitectural implementation, a unique entry inside the line-fill buffer. Second, we show three different variants of MEDUSA, which each have unique properties. Finally, we analyze potential attack targets for MEDUSA based on how real-world software uses implicit WC.

3.3.1 Leakage Analysis

To evaluate the practicality of MEDUSA, we first analyze the leakage of MEDUSA. This analysis includes the leakage source, the leakage pattern, how much control an attacker has over the leakage, and how much noise is in the leaked data. We first reduced the generated snippet, *i.e.*, we removed instructions as long as the leakage was still visible.

Leakage source. For the leakage source, TRANSYNTHER already provides an educated guess that the leakage source is the fill buffer. For MEDUSA, TRANSYNTHER reports a Pearson coefficient of $r_p = 0.99$ for the fill buffer, while the correlation for the other performance counters is not statistically significant. However, the only leaked value is the

	With memory barrier		Without memory barrier		>128-bit data	
	load	store	load	store	load	store
⊘	RIDL	RIDL	RIDL (ST)	RIDL (ST)	-	-
<✖>	-	Fallout (ST)	-	Fallout (ST)	-	**Medusa** / Fallout (ST)
🔒	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	**Medusa** / ZombieLoad
TAA	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad	ZombieLoad
PTE inversion	-	Fallout (CL, ST)	-	Fallout (CL, ST)	-	Fallout (CL, ST)
👆	ZombieLoad	ZombieLoad / Fallout (ST)	ZombieLoad	ZombieLoad / Fallout (ST)	ZombieLoad	ZombieLoad / Fallout (ST)
Attack(s)	ZombieLoad / RIDL	ZombieLoad / RIDL / Fallout (ST)	ZombieLoad / RIDL (ST)	ZombieLoad / RIDL (ST) / Fallout (ST)	ZombieLoad	**Medusa** / ZombieLoad / Fallout (ST)

Table 3.4: A comparison of MDS attacks in various variants and on different targets.

character written with `rep stosb`. Hence, in contrast to ZombieLoad [300], MEDUSA can only leak from a part of the line-fill buffer.

We additionally verify that using the publicly available proof-of-concept for ZombieLoad. Using this victim, we do not see any leakage when using MEDUSA, while we see a substantial leakage when using the ZombieLoad attack. We also used the public proof-of-concept for RIDL [356]. Interestingly, RIDL only works when reading data after a flush and a memory barrier. If either the flush or the memory barrier (*i.e.*, `cpuid` or `mfence`) is missing, we do not get any leakage.

In Section 3.3.1, we compare different victims and whether any variant of an MDS attack (ZombieLoad, RIDL, Fallout) or MEDUSA can leak data from the victims. While larger data than 128 bits, e.g., `rep mov`, can also be leaked with ZombieLoad (same and cross hyperthread) or Fallout (same hyperthread), MEDUSA only reveals data larger than 128 bits. Hence, while MEDUSA does not exploit any new data source, it targets precisely one type of victim, and there is no unrelated data from other processes.

WC and fill buffer. According to Intel, their microarchitectures use the line-fill buffer as WC buffers [173]. Thus, officially, WC can use ten line-fill-buffer entries [177]. Schwarz et al. [300] experimentally verified this for pre-Skylake microarchitectures but detected 12 line-fill-buffer entries since Skylake. We devised several experiments to analyze the line-fill buffer's WC-behavior for all memory types supported on x86_64.

While there is an explicit WC memory type, specific instructions always use WC independent of the underlying memory type, e.g., non-temporal stores. Depending on the CPU, Intel also documents that non-temporal loads (`MOVNTDQA`) may reduce the number of cache evictions by leveraging the WC buffer [177]. Recent Intel CPUs support fast-string operations via the `rep mov` and `rep sto` instructions [163, 177]. These instructions do not guarantee any order of the written data [177]. Hence, they can employ WC to reduce the number of write requests sent on the memory bus. We verified that with Medusa, we could leak the values both for explicit WC, *i.e.*, memory marked as WC, and implicit WC, *i.e.*, `MOVNTDQA`, `rep mov`, and `rep sto`. Hence, Medusa has the unique property among all MDS attacks that instruction types filter the leakage, *i.e.*, the amount of unrelated data is significantly less than in other attacks.

Leakage pattern. Figure 3.3 shows the leakage pattern for Medusa when copying a 256-byte buffer in the victim application using `rep mov` over the time of 10 s. We can see that while it is infeasible to leak all offsets in a 256-byte window with the same frequency, all offsets can be leaked.

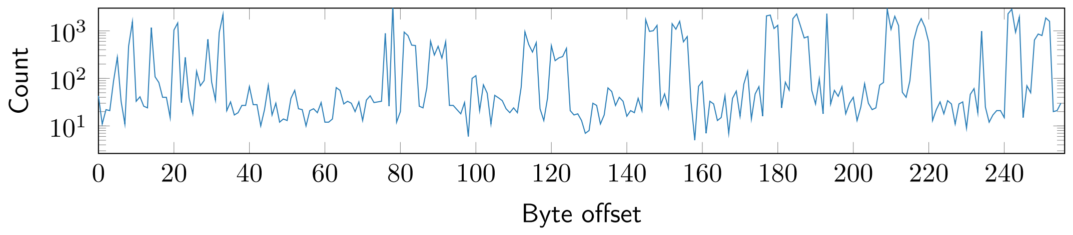


Figure 3.3: Leaking values with Medusa when copying a 256-byte buffer using `rep mov` shows an interesting pattern. While one can leak all bytes, certain offsets in the buffer have a much higher probability of leaking.

For the victim, we use a de Bruijn sequence of order three on an alphabet of size 26, *i.e.*, $B(26, 3)$, to groom the WC buffer (cf. Section 3.2.4). We continuously write this sequence to a dummy location using `rep mov`. The victim is running on the sibling CPU thread.

For the attacker, we always leak 3 bytes at a time by encoding every byte into a different array of 256 pages. As it is possible to compute on the full leaked values in the transient domain [300], we can reveal a 32-bit value, split it, and encode it to different arrays. Then we can match the recovered 3-byte value to the de Bruijn sequence used in the victim application. As every 3-byte value within the de Bruijn sequence is unique, this method allows us to analyze the leaked values pattern. Notably, we can always see strides of values that often occur in the leaked data, followed by strides that only happen rarely. Especially for the beginning of the buffer, the probability for leaking the first 32 bytes ($p =$67 %, $n =$10 000) is significantly higher than for leaking the second 32 bytes ($p =$33 %, $n =$10 000). We assume that the split of 32 B is due to the 32 B data-bus size on our test machine (i7-8650U). Hence, to transmit a WC-buffer entry over the shared data bus, both halves of the entry have to be transferred separately, and Medusa leaks either the first or second half. Data after the first cache line shows a different pattern. We can always see 16 B strides of values that occur often in the leaked data, followed by 16 B strides, which only happen rarely. Interestingly, this pattern does neither correlate with the bus size nor the WC buffer size. The leakage rate also increases after the first 64 bytes. At the time of writing, we do not know how to further analyze these effects; hence, we leave investigating this effect for future work.

Entry size for WC buffer. In the first experiment, we determine the size of an entry in the WC buffer. The basic idea is to detect that there are no available WC-buffer entries anymore. For this, we rely on the `L1D_PEND_MISS.FB_FULL` performance counter.

We execute an increasing number of non-temporal linear store instructions with a defined stride size in the experiment. Non-temporal stores ensure that the CPU uses WC for the stores. When the stride size exceeds the size of a WC-buffer entry, the CPU must allocate a new WC-buffer entry for every store. Hence, if we see that the WC buffer becomes a bottleneck, and the number of executed stores matches the number of fill-buffer entries, we know that the stride size equals the WC-buffer-entry size.

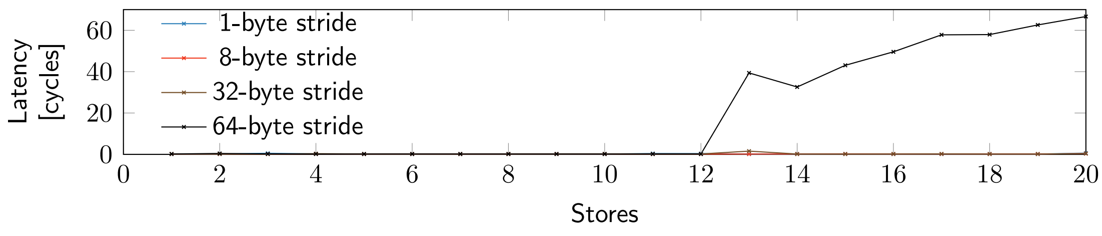


Figure 3.4: The number of cycles no fill-buffer entry is available. As there are 12 fill-buffer entries since Skylake [300] which are used as WC-buffer entries [173], the WC-buffer-entry has to be 64 bytes, *i.e.*, one cache line.

Figure 3.4 shows the results of this experiment. The performance counter reports the WC buffers' unavailability only at a stride size of 64 bytes and more than 12 stores. For smaller stride sizes, the WC buffer can combine the stores such that not every store requires its separate buffer entry.

3.3.2 Exploitation Methodology

In the following, we describe three different variants that allow to trigger MEDUSA.

Variant I: Cache indexing. In the first variant of MEDUSA, we rely on faulting loads located inside a cache line. Variant I exploits faulting loads on addresses that point *inside* a cache line (cf. Figure 3.5) to leak values from the WC buffer. The setup is similar to all Meltdown-type attacks, with a faulting load that transiently encodes the loaded data into a microarchitectural element. In contrast to existing attacks, the type of fault is not essential, but the faulting address's cache-line offset is. We verified Variant I with both non-canonical and supervisor addresses. On our test machine, an i7-8650U, the cache-line offset, *i.e.*, the least-significant 6 bits of the address, has to be at least 8, which is the maximum size for normal memory loads. However, the highest leakage rates are for offsets between 16 and 31. The common data bus has a width of 32 bytes. However, normal loads can only use 8, and AVX loads 16 bytes (128 bits). Consequently, offsets 16 to 31 are rarely used, as only AVX2 (256 bits) uses the shared data bus's full width. However, as WC's goal is to increase the throughput, implicit WC operations also try to leverage the entire common data bus. Hence, by using address offsets that index the upper half of the common data bus, Variant I leaks stale values of recent WC operations, e.g., `rep mov`, as well as AVX2 memory loads.

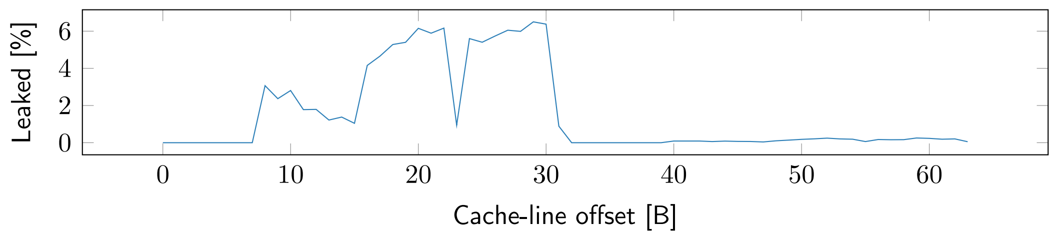


Figure 3.5: The cache-line offsets and how they contribute to the leakage for Medusa Variant I.

While at first, Variant I appears to be similar to MLPDS [157], ZombieLoad [300], or Fallout [61], it has distinctive properties. First, MLPDS requires either a faulting load spanning a cache line (64 B) or a faulting vector load that is larger than 64 bits [157]. For Variant I, neither of these requirements is necessary. In contrast, Variant I only works if the load is within one cache line. Loads spanning over two cache lines do not show data leakage (cf. Figure 3.5). Second, Variant I leaks data from the same CPU thread and the sibling thread, which is different from Fallout [61]. The leakage is limited to data stored using either `rep mov`, `rep sto`, or AVX2. In contrast to ZombieLoad or Fallout, Variant I of Medusa is agnostic to other data passing the store buffer or the fill buffer, since they never use the upper half of the shared data bus.

Variant II: Unaligned store-to-load forwarding. Faulting or assisting load that meets the "Unaligned Store-to-Load Forwarding" condition (similar to MSBDS) consistently leaks stale data. We have previously observed this behavior even across SMT.

	WB	WC	WT	UC
REP MOV	483 B/s, 99 %	299 B/s, 97 %	122 B/s, 43 %	0 B/s, 0 %
REP STO	656 B/s, 99 %	1960 B/s, 57 %	511 B/s, 49 %	471 B/s, 73 %
NT MOV	0.1 B/s, 99 %	0.1 B/s, 97 %	0.1 B/s, 43 %	0.1 B/s, 73 %

Table 3.5: Variant II performance for various memory types and victim operations.

Note that this is different from MSBDS, as MSBDS does not work across sibling threads. Here, we can leak the WC buffer data by creating an unaligned store-to-load forwarding condition on faulting or assisting load. Further, an attacker can control to leak which bytes of the WC buffer by combining various load sizes and the small store's offset. In our experiments, we managed to control the last 16 bytes of a WC buffer line by combining a 32-byte read 'ymmX' and iterating over various values for the store offset.

Variant III: Shadow `rep mov`. Variant III of MEDUSA exploits a microcode assist caused by a `rep mov` followed by a dependent faulting load. The `rep mov` instruction copies a single dummy byte to a destination address, which causes a fault, e.g., a non-canonical address. A subsequent load from the destination address leaks data from a stale or concurrent `rep mov`. The `rep mov` can either be on the same logical core before running MEDUSA, which leaks stale data of the previous `rep mov`. This leakage also works across privilege boundaries, *i.e.*, the stale `rep mov` data can also be from the kernel. Moreover, this attack also works for a concurrent `rep mov` on the sibling logical core across privilege boundaries.

As with Variant I, this variant has the property to only leak data of `rep mov`, `rep sto`, and AVX2 loads, which allows a targeted leakage of data used in such constructs. In contrast to Variant I, this variant is entirely address-agnostic and simplifies the recording of the leakage. However, since an attacker can not control the index of the leaked data it increases the post-processing complexity. Hence, as we can leak every byte of the victim buffer with a certain probability, the post-processing has to stitch together the leaked data, e.g., using the Domino technique presented by Schwarz et al. [300].

3.3.3 WC in Real-World Software

We analyzed real-world software to find occurrences of WC. We looked both for explicit WC, *i.e.*, WC memory defined through the PAT, and implicit WC is in the form of `rep mov` and `rep sto`.

Library	Version	O0	O1	O2	O3	Os
Botan	2.11.0	12	14	68	*137	188
Openssl	1.1.1c	12	23	29	*34	347
Wolfssl	4.1.0	1	7	*49	72	199
Bearssl	0.6	10	26	45	56	*213
Sodium	1.0.18	3	12	*12	13	49
Gcrypt	1.8.4	5	5	*7	11	168

Table 3.6: `rep mov` instruction within cryptographic libraries.

Userspace. As userspace application cannot directly change the memory type for a memory page, WC is mostly implicit in userspace. We analyzed when and how often GCC emits a `rep mov` sequence when compiling applications. We focussed mainly on potential attack targets, *i.e.*, an application that processes sensitive information. If GCC optimizes the application for code size (`-Os`), it emits the most `rep mov` instructions as `rep mov` is the smallest possible code sequence that can copy memory regions. Similarly, `rep sto` is the smallest code sequence to initialize memory with a defined value.

In addition to the compiler's implicit WC usages, we also found the explicit use of WC memory types in the userspace. Although implementation-specific, both OpenGL and Vulkan support memory buffers marked as WC. Memory buffers allocated as write-only buffers are likely to be assigned as WC memory by the driver.

Linux kernel. The Linux kernel also relies on `rep mov` to copy data. In contrast to userspace applications, the usage of `rep mov` is not to optimize the kernel binary for size. It is used independently of the used compilation flags, as the kernel generally does not use floating-point or SIMD operations. Hence, `rep mov` is the most efficient way to copy data. As there is a small startup penalty when using `rep mov`, only strings with a minimum length of 64 B use `rep mov` for a copy. For shorter strings, or if fast-string operations are not supported, the kernel falls back to a simple copy loop. We reverse-engineered the kernel binary for kernel 5.0.0 shipped with Ubuntu to analyze it for the usage of `rep mov`. We found 517 usages of `rep mov` in 374 functions in the binary. While many of the functions are only used once in the kernel setup phase (e.g., to copy and decompress parts of the kernel, set up EFI and several devices, initialize the architecture, or apply microcode updates), some of them are used regularly. These functions include, amongst others, `memcpy`, `memmove`, `copy_from_user`, and `copy_to_user`.

3.3.4 Leakage Performance of Medusa

We evaluated the performance of MEDUSA based on our proof-of-concept implementations.

Environments. We evaluated all variants of MEDUSA on our Intel and AMD CPUs mentioned before. All environments run Ubuntu with a recent 5.0 kernel version. For CPUs vulnerable to Meltdown, we enable the KPTI software mitigation. We successfully used all variants in all tested environments.

Performance. We consider the leakage rate and the false-positive rate when using MEDUSA on a colluding victim to evaluate the performance. This choice provides an upper bound for the leakage rates we can expect when using MEDUSA in a side-channel attack where the victim is not colluding. We started a victim application on one logical core, which leaks a known value. On the sibling hyperthread, we ran MEDUSA repeatedly for 2 s and recorded the correctly and incorrectly leaked values. With Variant 1, we achieve an average leakage rate of 0.19 kB/s ($n = 100$, $\sigma_{\bar{x}} = 0.0023$), with a false-positive rate of 47.7 % ($n = 100$, $\sigma_{\bar{x}} = 0.002$). For Variant 2, the leakage rate is on average 36.23 kB/s ($n = 100$, $\sigma_{\bar{x}} = 0.15$) with a false-positive rate of 0.559 % ($n = 100$, $\sigma_{\bar{x}} = 0.0005$). Finally, with Variant 3, we achieve an average leakage rate of 0.13 kB/s ($n = 100$, $\sigma_{\bar{x}} = 0.0016$) and a false-positive rate of 3.91 % ($n = 100$, $\sigma_{\bar{x}} = 0.0017$).

We have based these numbers on our unoptimized proof-of-concept implementation. Hence, these numbers cannot be taken as upper bounds for the leakage rate (and false-positive rate), as we expect that the leakage can be improved when improving the implementation.

Cross-VM covert channel. To evaluate the leakage rate of MEDUSA in the cross-VM scenario, we evaluate the performance of a cross-VM covert channel. While user applications can mount the covert channel, we focus on the cross-VM case as it is the most restricted scenario. For our setup, we use two co-located VMs running on an Intel Core i7-8650U running Ubuntu 18.04.3. Both VMs are running Ubuntu 18.04.3.

For the **sender**, we use a `rep mov` instruction, which continuously copies a 256-byte buffer containing the encoded data. We redundantly encode every 32-bit data packet by repeating it 32 times inside the buffer. Every 32-bit data packet consists of 8-bit data, an 8-bit checksum, a constant prefix, and a sequence number. The data-packet format resembles the setup from Schwarz et al. [300] to make the results comparable.

The **receiving** application leverages MEDUSA Variant 3 to leak victim data. Although the leaked data's redundancy reduces the speed, it increases the robustness, as any part of the leaked buffer contains the data. Moreover, due to the checksum, which we can already verify during the transient execution [300], we do not receive any unrelated data, making the receiver robust against any system noise.

We observed an average transmission rate of 14.3 B/s ($n = 1000$, $\sigma_{\bar{x}} = 0.56$) in the cross-VM scenario. In all cases, the transmission was error-free. Due to the encoding scheme's overhead, the performance is significantly slower than MEDUSA Variant 3 (cf. Section 3.3.4). We expect that more sophisticated encoding schemes, including error correction [233], can significantly improve the performance of the covert channel.

Leaking kernel data transfers. As discussed in Section 3.3.3, the Linux kernel also used `rep mov` for the internal data-transfer functions, including `memcpy`, `memmove`, `copy_from_user`, and `copy_to_user`.

Root password hash. As described by Van Schaik et al. [356], the unprivileged `passwd -S` command reads the contents of the user-inaccessible `/etc/shadow` file containing the password hashes of local users. They managed to leak 21 B in 24 h using the RIDL attack. Schwarz [296] showed that the same attack is more efficient with ZombieLoad by leaking 16 B in 1.25 min.

We also reproduced this attack using MEDUSA. While we can also leak the root password hash with MEDUSA, the leakage rate depends on the hash's leaking part. Due to the leakage pattern of MEDUSA, we always can leak blocks of the hash within 1 s, similarly to ZombieLoad, while for other blocks, it takes up to 1 h per byte, similarly to RIDL.

File I/O. Generally, MEDUSA can leak any data transfer between the kernel and the userspace, such as the files' contents when reading or writing them. We verified that we could leak the content by using a file with known contents. We continuously read the file from one application running on one hyperthread while running MEDUSA in a different userspace application on the sibling hyperthread. As the kernel handles every file read via the `read` syscall, the entire file content is copied from the kernel to the userspace victim application. On average, we were able to leak 12.3 B/s of correct values from the file.

Another case of data transfer is swapping. If the system copies application pages to or from the swap device, an adversary can potentially leak the data using MEDUSA.

3.4 Discussion

In this chapter, we performed an in-depth analysis of MDS attacks. We introduced a fuzzing-based analysis tool Transynther, which mutates the basic block of existing variants of Meltdown attacks to generate new subvariants. We analyzed selected CPUs using Transynther to better and found new MDS variants that only target fast string copies. Our findings proposed a new attack named Medusa, which leaks data from WC memory operations. Since Medusa uses only specific operations, it is more targeted. To demonstrate the effectiveness of Medusa, we ran several case studies. We demonstrated how one could recover information from kernel data transfers such as the root password hash or leak the content using a file with known contents. Further, using Medusa, we will show how to recover full RSA keys from OpenSSL by pooling leakages observed during key decoding, amplified using lattice techniques(§6.3).

We have designed the Transynther to find Meltdown-type vulnerabilities automatically. Other transient-execution attacks, such as Spectre-type attacks, are not in scope for Transynther. The reason is that Spectre attacks exploit the intentional, well-understood behavior of branch predictors. Spectre attacks can abuse several branch predictors [62], and the types of branch predictors are usually documented for every microarchitecture. Hence, we do not expect that Transynther would detect any new Spectre variants even when adapted for finding such attacks. Meltdown-type attacks, however, exploit CPU vulnerabilities that can be triggered in multiple different ways. Hence, as this paper has shown with Medusa, Transynther can discover new variants and potentially even help find Meltdown-type attacks on different platforms. In related work, Xiao et al. [378] analyze both Meltdown- and Spectre-type vulnerabilities in terms of speculation window, triggers, and different covert channels. They also rely on templates to build code snippets analyzed for vulnerabilities. Next, we discuss how Transynther can benefit from hardware simulation and information flow tracking (§3.4.1), and then we generalize the root cause for Meltdown-style vulnerabilities (§3.4.2).

3.4.1 Extending Transynther

Transynther works at the post-silicon stage in which we test an actual CPU rather than a pre-silicon schematic or simulator. Additionally, we do not have access to debug features that may expose microarchitectural components over a debug protocol like the JTAG. This ability of Transynther to perform black-box testing on a real product is one of our technique's benefits. It enables third-parties to do security testing of closed-source hardware, and our findings will directly translate to real-world security problems. Additionally, we could improve the accuracy of Transynther with access to open-source hardware or debug features. Such improvements may also enable the adaptation of information flow tracking for formal analysis and leakage quantification. In this section, we discuss some of these potentials in more detail:

Debug features. Modern CPUs support advanced post-silicon debug features that are generally not available to end-users. We have seen that previous vulnerabilities in the CPUs' firmware have allowed researchers to enable such debug features [101]. These debug features may allow access to internal registers that map to both off-core and core-private CPU resources such as cache and internal buffers.

TRANSYNTHER can benefit from access to such registers for both the grooming stage and the last stage of the meltdown-style attack, which currently uses a Flush+Reload sequence to observe the potential modified state of the cache. Instead, with access to such internal registers, we could potentially place arbitrary data into these internal components rather than relying on the architectural instructions for buffer grooming. One challenge of using architectural instructions for buffer grooming is isolating propagation of tracked data into a specific component, which we could have avoided if there was a mechanism to modify particular buffers' value. Additionally, we could replace the Flush+Reload sequence, which could suffer from noise, observing the cache state directly to get more reliable covert-channel feedback.

We expect these changes to improve the accuracy of TRANSYNTHER and further reveal leakages and understandings that we can not observe now by solely relying on architectural analysis. However, these hardware debugging is not possible with a high frequency, reducing the analysis speed.

Presilicon simulation and testing. Generally, engineers simulate the design of a system on an FPGA for pre-silicon testing [55]. Even if existing hardware debugging features do not support internal CPU components, we can modify the simulator to add accessibility and visibility to internal components. As a result, similar improvements as having access to debug features should be possible for TRANSYNTHER and with more visibility. However, pre-silicon simulation on an FPGA runs with a lower frequency than an actual CPU, which again reduces analysis speed and the number of tests we can execute. Additionally, transient execution attacks are sensitive to timings [247, 378], which may cause pre-silicon testing to experience some false positives or negatives compared to a real-world product.

In summary, there are different benefits and trade-offs in terms of real-world results, accuracy, and analysis speed with these approaches, including testing a final product (as in TRANSYNTHER), open-access testing, or pre-silicon testing. Therefore, we encourage the community to explore these options further for automated microarchitectural vulnerability testing.

Information flow tracking. In TRANSYNTHER, we use heuristics to observe if a placed value inside the microarchitecture can be leaked over the covert channel. However, with access to open-source hardware and pre-silicon testing, we will gain more visibility and accuracy. We can exploit this visibility to apply the state-of-the-art information flow tracking techniques for microarchitectural security testing [289]. For instance, we can track a specific value to see if it taints to various microarchitectural components before reaching a covert channel and mark them as relevant to the leakage source. This approach would allow us to provide automated explainability rather than manually analyzing the artifacts with performance counters, an alternative in TRANSYNTHER.

We may also reach tighter security boundaries by checking possible conditions where transient data falsely reach an architecturally-visible component. In contrast, currently, we only rely on a cache covert channel, which is not the only component for constructing covert channels. In a similar direction, speculative taint tracking (STT) proposes mitigation for Spectre by tracking the flow of annotated information from the software into the microarchitecture [388]. Although this work suggests mitigation for annotated secrets, one can combine similar taint tracking techniques with a fuzzing-based approach like TRANSYNTHER to find and analyze microarchitectural vulnerabilities without annotation.

Furthermore, Information flow tracking for automated testing of transient execution attacks creates the opportunity for automated impact analysis and classification of these vulnerabilities. Our work relies on end-to-end attack demonstration on cryptographic software to show the impact of these vulnerabilities in an empirical fashion. Demme et al. [84] suggested side-channel vulnerability factor (SVF) as a metric to quantify the amount of information leakage for side channels. A metric like SVF allows designers to choose better security and performance trade-offs based on the amount and type of leaked information. However, we are not aware of such metrics for transient execution attacks. With a reliable pre-silicon testing environment and formal information flow tracking, one may ultimately quantify the amount of leaked information and correlate it with the leakage source. Quantification is crucial when we design mitigations. For example, without formal quantification, we can not tell if eradicating a covert channel like cache from these attacks would be a reasonable defense since we are not aware of how much information one can leak with a new or existing vulnerability relying on an alternate covert channel [36].

3.4.2 Meltdown Root Cause Generalisation

We automatically generated thousands of different combinations using Transynther. Transynther was able to reproduce Meltdown [224], ZombieLoad [300], RIDL [356], Fallout [61], also known as Microarchitectural Store Buffer Data Sampling (MSBDS), Store-to-Leak (S2L) [297], Spectre v1.2 [203], and Microarchitectural Load Port Data Sampling (MLPDS) [157]. Furthermore, with Transynther, we synthesized multiple new, previously unknown variants to trigger these attacks. As a result, by analyzing the generated variants, we gained additional insights into Meltdown-type attacks. We identified that the root cause of all known Meltdown-type attacks is that an aborted load operation consumes any data that can be fetched first and provides them with dependent instructions.

General root cause. From the vast amount of results generated by Transynther, we can generalize the common root cause of known Meltdown attacks. As stated by Canella et al. [62], a faulting load is the cause of the leakage for all known Meltdown, where microcode assists are essentially microarchitectural faults [300]. In all attacks, we see the same behavior, that the faulting load does not stall and cannot solely return no data. Consequently, the faulting load transiently returns data that can be accessed immediately and where at least parts of the address match.

Source of leakage. The attack leaks the data from different microarchitectural elements depending on the implementation of data-forwarding checks and where the fault occurs. For example, ZombieLoad and Fallout exploit the same fault as the original Meltdown attack, and RIDL uses the same condition as Foreshadow. In RIDL and Foreshadow's case, it is the cleared present bit in the load target's page-table entry. If the L1 cache contains data with an address that matches the page-frame number, the load takes this value. This case is known as Foreshadow or Meltdown-P-L1 [62]. If this is not the case, e.g., because the page-frame number is 0 in the case of a NULL-pointer, the next possibility for data with partial address matches is the line-fill buffer. This case is known as RIDL or Meltdown-P-LFB [62]. The same principle applies to Meltdown, ZombieLoad, and Foreshadow, where the user-accessible-bit in the page-table entry is exploited. First, the CPU checks both the store buffer in parallel with the L1 data cache. If a store-buffer entry has a partial address match, the faulting load consumes this data, known as Fallout or Meltdown-US-SB [61]. Otherwise, if the cache can provide data with partially matching addresses, this is considered as Meltdown-US [62]. In case the L1 cache cannot satisfy the request due to a cache miss or a cache-line conflict, the line-fill buffer is queried, resulting in ZombieLoad [300] or Meltdown-US-LFB [62].

Hence, one of the insights from TRANSYNTHER is that the type of the fault is less important than where the fault occurs, *i.e.*, which microarchitectural element is the "closest" to the fault from which the faulting load can consume data.

Chapter 4

Controlled Instruction-Level Attacks on Enclaves

Recent Intel CPUs include software guard extensions (SGX) [164] to allow trusted execution of critical code in so-called *enclaves* on top of a potentially compromised OS. In Section 4.1, after giving an overview of SGX secure enclaves, we provide a characterization of demonstrated attacks in the literature against Intel SGX. In particular, the adversarial model of secure enclaves allows a system-level adversary to control high-precision interrupts. In Section 4.2, we propose a novel interrupt-driven attack, called CopyCat, that improve previous side-channel attacks against secure enclaves in terms of precision, spatial resolution, and scalability. Finally, in Section 4.3, we discuss the application of CopyCat, including a case study on how to use CopyCat to bypass previous mitigations. In general, CopyCat also enables a new avenue of side-channel analysis and cryptanalysis in the context of secure enclaves. We cover this aspect in Section 6.4, in which we perform an extensive study of single-trace and deterministic attacks against cryptographic implementations. As a result, we propose novel algorithmic attacks to perform single-trace key extraction that exploits subtle vulnerabilities in the latest versions of widely-used cryptographic libraries.

4.1 Attack's Characterization

Secure enclaves. A trusted execution environment (TEE) allows trusted execution of instructions on an untrusted CPU by leveraging hardware supported memory isolation semantics, cryptographic primitives, and root of trust. The assumption is that the hardware controls the root of trust. Thus, even system adversaries should not subvert the confidentiality and integrity of processes executed in a trusted environment (secure enclave).

Intel Software Guard Extensions (SGX) introduced support for user-mode enclaves with the Skylake generation [192]. SGX provides an additional set of instructions that enable the operating system (OS) and application developers to instantiate a secure enclave that they can only invoke through a standard contract. Dedicated `eenter` and `eexit` instructions switch the CPU in and out of "enclave mode". SGX enclaves are isolated at runtime in a memory area that is transparently encrypted and can be remotely attested by the CPU. After correctly instantiating and validating the enclave module, the hardware guarantees the enclave's trusted execution. The OS is not allowed to influence the execution of enclave instructions and observe the enclave at runtime. However, recent microarchitectural attacks show how various microarchitectural properties can invalidate these assumptions [82, 300, 348].

The main goal of SGX is to protect runtime data and computation from the system and physical adversaries. SGX must remain secure in the presence of malicious OS; thus, modification of OS resources for the facilitation of side-channel attacks is relevant and within the considered threat model. Intel declared microarchitectural leakages out of scope for SGX, thus pushing the burden of writing leakage-free constant-time code onto enclave developers.

4.1.1 Microarchitectural Contention

In the past years, we have seen a continuous stream of software-based side-channel attacks [15, 21, 104, 115, 244, 384, 385]. The first category of microarchitectural timing attacks commonly abuses optimizations in modern CPUs, where a secret-dependent state is accumulated in various microarchitectural buffers during the victim's execution. If these buffers are not flushed before a context switch to an attacker domain, the attacker can reconstruct victim secrets by observing the attacker's timing variations. The success of these attacks critically relies on subtle timing differences, making them inherently non-deterministic and prone to measurement noise [115]. Usually, a common way to eliminate this class of stateful attacks is to isolate leaky microarchitectural resources [87, 225, 337, 379].

Notably, while the CPU always safeguards the confidentiality and integrity of enclaved execution, traditionally privileged OS software remains in charge of availability concerns. SGX enclaves live in the virtual address space of a conventional, userspace process. Enclave page-table mappings are verified but remain under the explicit control of the untrusted OS. This design allows for demand-paging and oversubscription of the physically available encrypted memory. The CPU may cache recent address translations in an internal translation lookaside buffer (TLB), which is flushed by the CPU on every enclave transition. When delivering asynchronous interrupts or exceptions, the CPU takes care to save and scrub CPU registers before exiting the enclave securely. The software can subsequently re-enter through the `eresume` instruction. Furthermore, in case of a page-fault event, the CPU clears the lower bits representing the page offset in the reported address to ensure that the OS can only observe enclave memory accesses at a 4 KiB page-level granularity.

	Attack	Code/Data	Granularity	Noise
μ-arch contention	DRAM row buffer conflicts [360]	Code + data	✘ Low (1-8 KiB)	✘ High
	Prime+Probe cache conflicts [49, 140, 245, 303]	Code + data	✘ Med (64-512 B cache line/set)	~ Med
	Read-after-write false dependencies [244]	Data	✔ High (4 B)	✘ High
	Branch prediction history buffers [104, 153, 218]	Code	✔ High (branch instruction)	~ Low
	Interrupt latency [352]	Code + data	✔ High (instruction latency class)	✘ High
	Port contention [15]	Code	✔ High (μ-op execution port)	✘ High
Ctrl channel	Page faults [380] and page table A/D bits [353, 360]	Code + data	✘ Low (4 KiB)	✔ Deterministic
	IA-32 segmentation faults [138]	Code + data	✘ Low/high (4 KiB; 1 B for enclaves $\leq$ 1 MiB)	✔ Deterministic
	Page table Flush+Reload [353]	Code + data	✘ Low (32 KiB)	~ Low
	CopyCat	**Code**	**✔ High (instruction)**	**✔ Deterministic**

Table 4.1: Characterization of demonstrated Intel SGX microarchitectural side channels (top) and controlled channels (bottom). Our novel CopyCat technique is highlighted at the bottom and combines noise-free interrupt counting measurements with deterministic page table accesses to reconstruct enclave-private control flow at a maximal, instruction-level granularity.

While Intel SGX provides strong architectural isolation, several studies have highlighted that enclave's secrets may still leak through side-channel analysis. Section 4.1.1 summarizes how all previously demonstrated side-channel attacks fall into two categories:[1]

- microarchitectural timing attacks, which may achieve a high granularity but are inherently prone to measurement noise, and
- fully deterministic controlled-channel attacks that only offer a relatively coarse-grained 4 KiB page-level granularity.

CopyCat proposes the only generally applicable controlled-channel attack that is both fully deterministic and offers a maximal, instruction-level granularity.

Microarchitectural timing side-channel attacks exploit the fact that various resources, such as caches [49, 140, 245, 303], DRAM row buffers [360], branch predictors [104, 153, 218], dependency resolution logic [244], or execution ports [15] are competitively shared between sibling CPU threads or not flushed when exiting the enclave. This contention causes measurable timing differences in the attacker domain, allowing the attacker to

[1] Transient-execution attacks [300, 348, 349] are orthogonal to metadata leakage through side channels and require recovery of the trusted computing base through complementary microcode and compiler mitigations.

infer the enclave's private control flow or data access pattern with varying degrees of granularity. In the context of a TEE such as Intel SGX, attackers can mount such attacks with less noise and improved resolution because the adversary controls the OS.

In particular, one line of work has developed interrupt-driven attacks [140, 218, 245, 351] that rely on frequent enclave preemption to sample side-channel measurements at an improved temporal resolution. This technique has been demonstrated to amplify side-channel leakage from the cache [245], the branch target buffer [218], and the directional branch predictor [153]. Researchers have demonstrated similar techniques on ARM TrustZone [287]. Nemesis [352] showed that while single stepping, the response time to service an interrupt may reveal which instruction pipeline is executing. The SGX-Step framework [351] has been leveraged in several other microarchitectural attacks [14, 153, 300, 348, 349, 352] to reliably single-step enclaves at a *maximal* temporal resolution by means of precise and short timer interrupt intervals.

4.1.2 Controlled-Channel Attacks

Orthogonal to the first class of microarchitectural timing attacks, recent research on *controlled-channel* attacks [138, 353, 360, 380] has abused the processor's privileged software interface to extract fully deterministic, noise-free side-channel access patterns from enclave applications. While the operating system (OS) was traditionally not under the attacker's control, this assumption fundamentally changed with the rise of trusted execution environments (TEEs), such as Intel SGX. Prior work [353, 380] has identified page-table accesses and faults as privileged interfaces that can be exploited as no-noise controlled channels to deterministically reveal enclave memory accesses at a 4 KiB page-level granularity. The paging channel has drawn considerable research attention since it abuses the x86 processor architecture's intrinsic property without relying on microarchitectural states. In particular, controlled-channel attacks have proven to be challenging to mitigate in a principled way, in spite of numerous defense proposals [69, 72, 266, 290, 311, 312, 326].

Xu et al. [380] first showed how privileged adversaries could revoke access rights on a specific enclave page and get a deterministic notification using a page-fault signal when the enclave next accesses that page. They demonstrated several attacks on non-cryptographic applications by observing that page-fault sequences uniquely identify specific points in the victim's execution. Subsequent work [353, 360] developed stealthier techniques to extract the same information without provoking page faults. These attacks interrupt the victim enclave to flush the TLB forcefully and provoke page-table walks, which can later be reconstructed through "accessed" and "dirty" attributes or cache timing differences for untrusted page-table entries. Finally, Gyselinck et al. [138] demonstrated an alternative controlled-channel attack that abuses legacy IA32 segmentation faults. Their attack offers an improved, byte-level granularity in the first MiB of the enclave address space, but only for the unusual case of a 32-bit enclave. Recent microcode updates have fixed this behavior.

With CopyCat, we contribute an improved attack technique to refine the resolution of existing controlled channels by precisely counting the number of executed enclave instructions between successive page accesses. Prior work has similarly suggested an additional temporal dimension for the paging channel by using interrupts to reconstruct `strlen` loop iterations [350, 351], or by logging noisy wall-clock time [360] for page-access events to improve stealthiness and reduce the number of TLB flushes. Recent work [200] on enclave control flow obfuscation furthermore investigated using single-stepping in an SGX simulator to identify software versions in an emulated enclave debug environment probabilistically. This work shares the same core idea with CopyCat but does not implement actual instruction counting attacks or provide a deterministic single-stepping interrupt primitive outside of a simulator. In contrast to these specialized cases, CopyCat explicitly recognizes instruction counting as a practical and generically applicable attack primitive that can deterministically capture the execution trace within a single enclave code page.

4.2 CopyCat: Instruction-Counting Side Channel

As already discussed in this chapter, the adversarial model presented by trusted execution environments (TEEs) has prompted researchers to investigate unusual attack vectors. One incredibly powerful class of *controlled-channel* attacks abuses page-table modifications to reliably track enclave memory accesses at a page-level granularity. In contrast to the noisy microarchitectural timing leakage, deterministic controlled-channel attacks abuse indispensable architectural interfaces. These attacks cannot be mitigated by tweaking microarchitectural resources.

We propose an innovative controlled-channel attack, named CopyCat, that deterministically counts the number of instructions executed *within* a single enclave code page. We show that combining the instruction counts harvested by CopyCat with traditional, coarse-grained page-level leakage allows the accurate reconstruction of enclave control flow at a *maximal* instruction-level granularity. CopyCat can identify intra-page and intra-cache line branch decisions that ultimately may only differ in a single instruction, underscoring that even extremely subtle control flow deviations can leak secrets from secure enclaves. We demonstrate the improved resolution and practicality of CopyCat on Intel SGX.

4.2.1 Introducing CopyCat

This section shows that the resolution of deterministic controlled-channel attacks extends well beyond the relatively coarse-grained 4 KiB page-level granularity. We introduce CopyCat, an innovative *interrupt-counting* channel that can precisely reconstruct the intra-page control flow of a secure enclave at a maximal, instruction-level granularity. Our attack leverages the SGX-Step [351] framework to forcibly step into a victim enclave code exactly one instruction at a time. While high-frequency timer interrupts have previously been leveraged to boost microarchitectural timing attacks [140, 153, 218, 245, 352], we exploit the architectural interrupt interface itself as a deterministic controlled channel. In short, our attacks rely on the critical observation that merely counting the number of times a victim enclave can be interrupted directly reveals the number of executed instructions.

Our attacks rely on the critical observation that interrupts can force the enclave to advance exactly one instruction at a time. Hence, merely counting the number of steps reveals the number of instructions executed in the victim enclave. We show that combining our fine-grained interrupt-based counting technique with traditional, coarse-grained page-table access patterns [353, 360] as a secondary oracle allows us to construct highly effective and deterministic attacks that track enclave control flow at a maximal, instruction-level granularity. Crucially, the improved temporal dimension of CopyCat overcomes the spatial resolution limitation of prior controlled-channel attacks, invalidating a fundamental assumption in some previous defenses [174, 312] that presumes that adversaries can only deterministically monitor enclave memory accesses at a coarse-grained 4 KiB granularity. Furthermore, in contrast to previous high-resolution SGX side channels [15, 218, 244, 245, 352] that rely on timing differences from contention in some shared microarchitectural state, CopyCat cannot be transparently mitigated by isolating microarchitectural resources.

This section introduces the adversary model and explains how a deterministic single-stepping interrupt primitive for SGX enclaves can be built before illustrating the basic principle behind CopyCat through toy examples.

Attacker model. We assume the standard Intel SGX root adversary model with full control over the untrusted OS [164]. SGX's sharp threat model is justified, for instance, by considering untrusted cloud providers under the jurisdiction of foreign states or end-users with an incentive to break DRM technology running on their device. Following prior work, we assume a remote, software-only adversary who has compromised the untrusted OS, allowing the x86 APIC timer device to be configured to precisely interrupt the enclave [140, 218, 245, 351] and modify page-table entries to learn enclaved memory accessed at a 4 KiB granularity [312, 353, 380]. Like previous attacks, we further assume knowledge of the victim application, either through source code or the application binary. We assume the enclave code is free from memory-safety vulnerabilities [350], and the Intel SGX platform is updated against transient-execution attacks [300, 348].

The adversary's goal is to learn fine-grained control-flow decisions in the victim enclave. In contrast to noisy microarchitectural side channels [15, 49, 218, 244, 245, 352], we can also target victims who process a secret only *once* in a single run (as is the case in key generation) and hence victims who cannot be tricked to perform computations on the same secret multiple times repeatedly. Crucially, in contrast to prior controlled-channel attacks [353, 380], CopyCat offers intra-page granularity, and we assume that conditional control flow blocks within the victim's enclave are aligned "to exist entirely within a single page" as officially recommended by Intel [174].

4.2.2 Building the Interrupt Primitive

Debug features like the x86 single-step trap flag are explicitly disabled by the Intel SGX design [164] while in enclave mode. Recent research, however, has demonstrated that root adversaries may abuse APIC timer interrupts to pause a victim enclave at fixed time intervals forcibly. We build our interrupt primitive on top of the open-source SGX-Step [351] framework, which offers a maximal temporal resolution by reliably interrupting the victim enclave at most one instruction at a time. SGX-Step comes as a Linux kernel driver and runtime library to configure APIC timer interrupts and untrusted page-table entries directly from userspace.

Deterministic single-stepping. We first choose a suitable value for the platform-specific `SGX_STEP_TIMER_INTERVAL` parameter using the SGX-Step benchmark tool on our target CPU. This value ensures that the victim enclave always executes at most one instruction at a time. Previous studies [153, 351, 352] have reported reliable single-stepping results with SGX-Step for enclaves with several hundred thousand instructions where in the vast majority of cases ($> 97\%$) the timer interrupt arrives within the *first* enclave instruction after `eresume`, *i.e.*, single-step, and in *all* other cases the interrupt arrives within `eresume` itself, *i.e.*, zero-step before an enclave instruction is ever executed. Furthermore, zero-step events can be filtered out by observing that the "accessed" bit in the untrusted page-table entry mapping the enclave code page is only ever set by the CPU when the interrupt arrived after `eresume`, and the enclave instruction has indeed been retired [352]. Hence, to achieve noiseless and deterministic single-stepping for revealing code and data access at an instruction-level granularity, we rely on the observation that an adequately configured timer *never* causes a multi-step. We then discard any zero-step events by querying the "accessed" bit in the untrusted page-table entry mapping the current enclave code page. The experimental evaluation in Section 6.4 confirms that our single-stepping interrupt primitive indeed behaves fully deterministically when using CopyCat to count several millions of enclave instructions.

Before entering the single-stepping mode, we first use a coarse-grained page-fault state machine to advance the enclaved execution to a specific function invocation on the targeted code page. Such page-fault sequences have been shown to locate specific execution points in massive binaries uniquely [312, 365, 380]. Once we discover the particular code page of interest, CopyCat starts counting instructions until detecting the next code or data page access to reveal instruction-level control flow.

We will clarify further how we narrow down the attack trace to a target function. In summary, we count the number of instructions between page visits. We use the paging channel as a secondary oracle to group instruction counts, which are not correlated to binary size. Following prior research [380], we first use a coarse page-fault sequence state machine to uniquely detect the target code page's start containing the secret-dependent branch. We then switch to single-stepping mode. Compilers in practice generate code with different page accesses at different instruction offsets in both branches for various reasons (data/stack accesses, subroutine calls). As an optimization, we first use a coarse-grained page-fault state machine to efficiently advance the enclaved execution to the targeted code page before switching to single-stepping with CopyCat to reveal instruction-level control flow within the code page of interest.

Effects of macro fusion. Interestingly, we found that we can use CopyCat to study a microarchitectural optimization in recent Intel Core CPUs, referred to as *macro fusion* [163, 372]. The idea behind this optimization technique is to combine specific adjacent instruction pairs in the front-end into a single micro-op that executes with a single dispatch and hence frees up space in the CPU pipeline.

Intel documents that fusion only takes place for some well-defined compare-and-branch instruction pairs [163, §3.4.2.2], which are additionally not split on a cache line boundary [163, §2.4.2.1]. We experimentally found that for fusible instruction pairs, CopyCat consistently counts *one* interrupt only, even though the enclave-private program counter has been advanced with *two* assembly instructions forming the fused pair. Our experimental observations on Kaby Lake confirm Intel's documented limitations, e.g., `test;jo` can be fused (interrupted once) but not `cmp;jo` (interrupted twice); and fusible pairs that are split across an exact cache line boundary are not fused (interrupted twice). Importantly, we found that macro fusion does *not* impact the reliability of CopyCat as a deterministic attack primitive. In all of our attacks, we consistently observed that macro fusion depends solely on the architectural program state, *i.e.*, opcode types, and their alignments. Hence, a given code path always results in the same deterministic number of interrupts.

To the best of our knowledge, CopyCat contributes the first methodology to research and reverse-engineer macro fusion optimizations in Intel CPUs independently. While our observations confirm that macro fusion behaves as specified, we consider a precise understanding of the macro fusion of particular importance for compile-time hardening techniques that balance conditional code paths).

4.2.3 Instruction-Level Page Access Traces

Leakage model. CopyCat complements the coarse-grained 4 KiB spatial resolution of previous page fault-driven attacks with a fully deterministic temporal dimension. By interrupting after every instruction and querying page-table "accessed" bits, CopyCat adversaries obtain an instruction-granular trace of page visits performed by the enclave. This trace may reveal private branch decisions whenever a secret-dependent execution path does not access the same set of code and data pages at every instruction offset in both branches. Importantly, even when both execution paths access the same sequence of code and data pages, and hence remain indistinguishable for a traditional page-fault adversary [380], we show below that compilers may in practice still emit unbalanced instruction counts between page accesses in both branches.

If/Else statement. Conditional branches are pervasive in all applications [140, 149, 218, 380], but even side-channel hardened cryptographic software may assume that carefully aligned if/else statements or tight loops cannot be reliably reconstructed (§6.4). Figure 4.1 provides a minimal example of an if statement that has been hardened using a balancing else branch, e.g., as in the Montgomery Ladder algorithm. The corresponding assembly code, as compiled by gcc, only differs in a single x86 instruction that can fit entirely within the same page and cache line. This 'if' branch is hence indistinguishable for a page-fault or cache adversary. While finer-grained, branch prediction side channels may still reconstruct the branch outcome, these attacks typically require several victim runs. They can also be trivially addressed by flushing the branch predictor on an enclave exit.

Figure 4.1 illustrates how CopyCat can deterministically reconstruct the branch outcome merely by counting the number of instructions executed on the P_0 code page containing the 'if' branch before control flow is eventually transferred to the P_1 code page containing the add function, as revealed by probing the "accessed" bit in the corresponding page-table entry. The example furthermore highlights that even if all of the code were to fit on a single code page $P_0 = P_1$, CopyCat adversaries could still distinguish both branches

```
if (c == 0){ r = add(r, d); } else { r = add(r, s); }
```

```
 test %eax,%eax
 je 1f
 mov %edx,%esi
1:
 call add
 mov %eax,-0xc(%rbp)
```

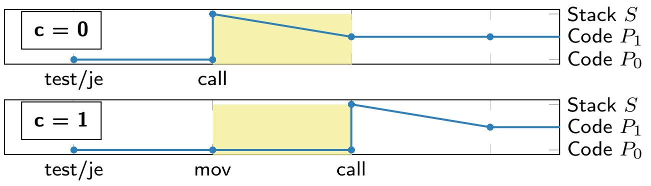


Figure 4.1: Balanced if/else statement (top), compiled to assembly (left). Precise page-aligned, intra-cache line conditional control flow can be deterministically reconstructed with instruction-granular CopyCat page access traces (right).

by comparing the relative position of the data access to the stack page S performed by the `call` instruction. In particular, while traditional page-fault adversaries always see the same *page fault sequence* (P_0, S, P_1), independent of the secret, CopyCat enriches this information with precise instruction counts, resulting in distinguishable *instruction-level page access traces* (P_0, P_0, S, P_1) vs. (P_0, P_0, P_0, S, P_1).

Switch-Case statement. As a further example, Figure 4.2 illustrates precise control-flow recovery in a switch-case statement. The code blocks again fall entirely within a single page and cache line, and where the code access the same data in every case. While traditional page-fault adversaries always observe an identical, input-independent access sequence to the code and data pages, and the tight arrangement of conditional jumps poses a considerable challenge for branch prediction adversaries [218], CopyCat deterministically reveals the complete control flow through the *relative* position of the data access in the instruction-granular page access traces.

4.3 The Effectiveness of CopyCat

CopyCat interrupts a victim enclave precisely one instruction at a time and relies on a secondary page-table oracle to assign a spatial resolution to each instruction-granular observation. Thus, our attack is only useful when the victim code contains a secret-dependent branch that accesses a different code or data page at the same instruction offset in both execution paths. In contrast to previous controlled-channel attacks [312, 353, 380], our notion of instruction-granular page access traces allows the *sequence* of code and data page visits in both branches to be identical.

We practically only need the target application to access a “marker” page at a different relative instruction offset in the secret-dependent execution path. We found that in practice,

```
switch(c)
{
  case 0:
    r = 0xbeef;
    break;
  case 1:
    r = 0xcafe;
    break;
  default:
    r = 0;
}
```

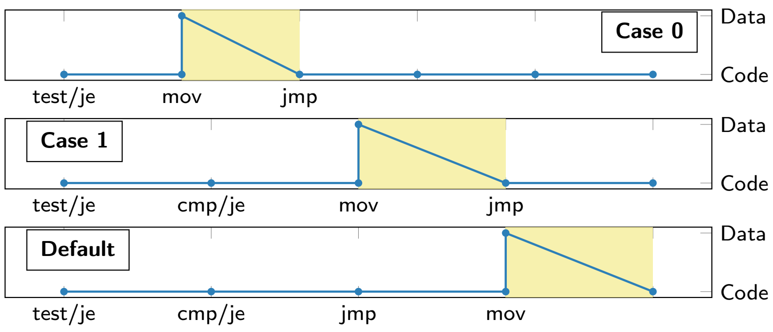


Figure 4.2: Conditional data assignments in a page-aligned switch statement (left) deterministically leak through their relative positions in the precise, instruction-granular page access traces extracted by CopyCat (right).

compilers generate code with different page accesses at different instruction offsets in both branches for a variety of reasons, including data or stack accesses, arithmetic operations, and subroutine calls. To highlight the importance of CopyCat for non-cryptographic applications, we employ its improved resolution to defeat a state-of-the-art compiler defense [149] against branch predictor leakage. This demonstration again shows that CopyCat changes the attack landscape and requires orthogonal mitigations compared to microarchitectural side channels.

4.3.1 Branch Shadow-Resistant Code

Listing 2 provides an elementary example function with secret-dependent branches. We provide the corresponding assembly output in Listing 3, as produced by the LLVM-based, open-source compiler mitigation pass [149] against branch shadowing attacks, described in Section 4.3.2. We enabled both rewriting of conditional branches via the trampoline area and protection against timing attacks via dummy instruction balancing by passing the `-mllvm -x86-branch-conversion` and `-mllvm -x86-bc-dummy-instr` options. Note that the open-source release has not integrated the randomizer. All code blocks on the trampoline area would still have to be randomly re-shuffled at runtime to protect against branch-shadowing attacks. For sufficient entropy, trampoline areas have to be larger than 4 KiB [149], and hence the trampoline will occupy at least one separate page.

We reveal control flow in the instrumented code of Listing 3 using CopyCat as follows. In the case where the secret-dependent 'if' condition is true, the indirect branch at line 20 will execute the single-instruction `jmp_if` block on the trampoline page, followed by 4 instructions on the instrumented code page, totaling 5 instructions before reaching

Listing 2 Sample code snippet with conditional branching.

```
void my_func(int a) {
    if (a  != 0) block1++; else block2++;
    block3++;
}
```

the end_if marker. In contrast, if the 'if' condition is false, the indirect branch at line 20 will transfer to the skip_if block on the trampoline page, totaling 4 instructions before eventually reaching the end_if marker back on the instrumented code page. Similar unbalanced instruction counts follow for the else block.

We experimentally verified that COPYCAT adversaries could deterministically learn the if condition by merely counting instructions and observing page accesses. Moreover, because the dummy instructions do not result in exactly balanced instruction counts, as explained above, merely counting the total amount of executed instructions even suffices in itself without having to distinguish accesses to the trampoline page.

4.3.2 Defeating Branch Shadowing Defenses

Lee et al. [218] first proposed Zigzagger, an automated compile-time approach to defend against branch-shadowing attacks by rewriting conditional branches as cmov and a tight trampoline sequence of unconditional jump instructions. However, their compiler transformation's security relies on the trampoline sequences being non-interruptible Previously, researchers have demonstrated several proof-of-concept attacks on Zigzagger using precise interrupt capabilities [138, 351, 352]. In response, Hosseinzadeh et al. [149] proposed improved compiler mitigation that employs runtime randomization. This mitigation dynamically shuffles jump blocks on the trampoline area. As a result, it effectively hides branch targets and making branch shadowing attacks probabilistically infeasible. Figure 4.3 illustrates how this mitigation redirects conditional branches through randomized jump locations ① on the trampoline page while ensuring that the program always executes all jumps ② outside of the trampoline in the same order. Finally, to protect against timing attacks, the trampoline code is explicitly balanced with dummy instructions ③ to compensate for skipped blocks in the instrumented code.

Case-study attack. We evaluated COPYCAT on the open-source[2] release of the compiler hardening scheme [149] based on LLVM 6.0. We refer to Section 4.3.1 for the

[2]Branch shadowing mitigation: https://github.com/SSGAalto/sgx-branch-shadowing-mitigation

Listing 3 Hardened assembly output, corresponding to the source code in Listing 2, as produced by the open-source branch shadowing mitigation LLVM compiler pass.

```
           jmp    my_func /***  BEGIN TRAMPOLINE ***/
jmp_done:  jmp    done
jmp_done2: jmp    done
skip_else: add    $0x0,%r13b    # compensating dummy
           lea    jmp_done2(%rip),%r15
           jmp    end_else
jmp_else:  jmp    else
skip_if:   add    $0x0,%r13b    # compensating dummy
           add    $0x0,%r13b    # compensating dummy
           lea    jmp_else(%rip),%r15
           jmp    end_if
jmp_if:    jmp    if      /***  END TRAMPOLINE ***/
my_func:   push   %rbp
           mov    %rsp,%rbp
           mov    %edi,-0x4(%rbp)
           cmpl   $0x0,-0x4(%rbp)
           lea    jmp_if(%rip),%r15
           lea    skip_if(%rip),%r13
           cmove  %r13,%r15
           jmp    *%r15
if:        mov    block1(%rip),%eax
           add    $0x1,%eax
           mov    %eax,block1(%rip)
           lea    skip_else(%rip),%r15
end_if:    jmp    *%r15
else:      mov    block2(%rip),%eax
           add    $0x1,%eax
           mov    %eax,block2(%rip)
           lea    jmp_done(%rip),%r15
end_else:  jmp    *%r15
done:      mov    block3(%rip),%eax
           add    $0x1,%eax
           mov    %eax,block3(%rip)
           pop    %rbp
           ret
```

full assembly output of a minimal C example program. First, we found that the dummy instruction balancing pass is not always entirely accurate and may result in execution paths that differ slightly by one or two instructions (cf. Section 4.3.1). Crucially, while such subtle deviations would indeed very likely not be exploitable through timing, as initially envisioned by the mitigation, we experimentally validated that CopyCat can deterministically distinguish the unbalanced paths. Second, even when the code paths are perfectly balanced, Figure 4.3 illustrates that merely counting the *number* of instructions

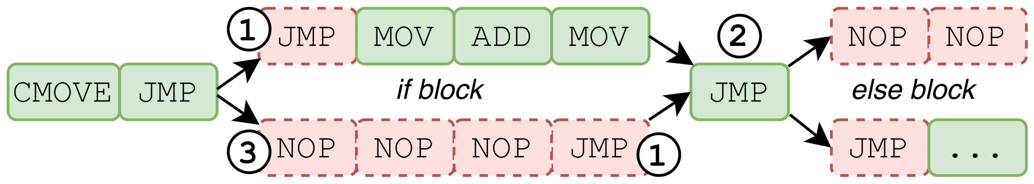


Figure 4.3: Compiler mitigation [149] for branch prediction side channels. (1) conditional branches are redirected through a randomized jump location on a trampoline page; (2) compensating dummy instructions are executed on the trampoline page to hide timing differences; (3) jumps outside of the trampoline area always performed in the same order. CopyCat reveals control flow via the number of instructions executed on the trampoline page (red, dashed).

executed on the trampoline page deterministically reveals whether the victim is running balancing dummy code in a trampoline block or the actual if the block on the instrumented code page. Note that the compiler carefully maintains a constant jump order when moving back and forth between the trampoline area and the instrumented code. The compiler ensures that the execution remains oblivious to classical page-fault adversaries [312, 380] who will always observe the same *sequence* of pages regardless of the executed code blocks.

4.4 Discussion

Our works show that deterministic controlled-channel adversaries are not restricted to observing enclave memory accesses at the level of coarse-grained 4 KiB pages, but can also precisely reconstruct intra-page control flow at a maximal, instruction-level granularity. We demonstrated the practicality and improved resolution of CopyCat by discovering highly dangerous single-trace key extraction attacks in several real-world, side-channel hardened cryptographic libraries. In contrast to known microarchitectural attacks, the more fundamental threat of deterministic controlled-channel leakage cannot be dealt with by merely flushing or partitioning microarchitectural state. Instead, it requires research into more principled solutions.

Comparison to branch prediction leakage. Section 4.1.1 identified branch prediction side channels [104, 153, 218] as an alternative attack vector to spy on enclave control flow at an instruction-level granularity with reasonable accuracy. In contrast to CopyCat, however, microarchitectural leakage from branch predictors is inherently noisy and typically requires multiple runs of the victim enclave, ruling out this class of side channels to perform noiseless single-trace attacks on key generation algorithms that we will present in Section 6.4.4. Furthermore, in contrast to the architectural interrupt and paging interfaces exploited by CopyCat, branch prediction side-channel leakage can be eradicated relatively straightforwardly by flushing branch history buffers when exiting the enclave, similar to the microcode updates Intel already distributed to flush branch predictors on enclave entry in response to Spectre threats [68]. Section 4.3.2 further highlighted the complementary aspects of interrupt counting and branch prediction leakage. We showed that CopyCat defeats state-of-the-art compiler defenses that harden code against branch prediction side channels [149].

In addition to the deterministic characteristic, CopyCat is significantly easier to scale and replicate, considering that branch predictors feature an intricate design that changes from one microarchitecture to another. BranchScope [104], for instance, relies on finding a heuristic through reverse engineering to probe a specific branch. This heuristic is dependent on

- the state of other components like global and tournament predictors; and
- the exact binary layout of the victim program.

Previous attacks focus on distinguishing one or a small number of branches. We believe that replicating BranchScope to probe multiple branches across various targets would be challenging and may even be practically infeasible. CopyCat, in contrast, is much easier to replicate, and we will show in Section 6.4 that our attack scales to probing the entire execution path in a single run.

Automation opportunities. The case-study attacks presented earlier relied on careful manual inspection of the victim enclave source code and binary layout to identify vulnerable secret-dependent code patterns. Similarly, we performed the CopyCat-based cryptanalysis in Section 6.4 on manual analysis of application code and binary. We expect that dynamic analysis and symbolic execution approaches could further improve our attacks' effectiveness and increase confidence for defenders by automating the discovery of vulnerable code patterns [359, 368] and possibly even the synthesis of proof-of-concept exploitation code. While the requirements for vulnerable code patterns are relatively clear-cut, as described above, we expect that it may be particularly challenging to track the propagation of secrets and distinguish between automatically non-secret and secret-dependent control flows [38].

Chapter 5

Timing Analysis of Physically-isolated Elements

In this chapter, we look into a popular cryptographic-coprocessor, the trusted platform module (TPM). We show that although physical isolation promises stronger security guarantees, it is still susceptible to side-channel attacks. Notably, the tight integration of these security chips into the system facilitates the exploitation of side channels. Section 5.1 provides some background information regarding TPMs, their implementation and previous security issues. Then, Section 5.2 discuss our part of our contribution in TPM-Fail, which shows that precise timing analysis of these devices reveals critical security vulnerabilities. Finally, we summarize our findings in Section 5.3

5.1 Trusted Platform Module

As we mentioned throughout this dissertation, hardware support for trusted computing has been proposed based on trusted execution environments (TEE) and secure elements such as the Trusted Platform Module (TPM) [242]. Trusted Platform Module (TPM) serves as a hardware-based root of trust that protects cryptographic keys from the privileged system and physical adversaries. Computer manufacturers have been deploying TPMs on desktop workstations, laptops, and servers for over a decade. With a TPM device attached to the computer, computer manufacturers can execute the root of trust in a separate hardened cryptographic core, preventing even a fully compromised OS from revealing credentials or keys to adversaries. TPM 2.0, the latest standard, is deployed in almost all modern computers and is required by some core security services [238]. TPM 2.0 supports multiple signature schemes based on elliptic curves, which helps applications to benefit from the state-of-the-art and more efficient signing operations for remote attestation [342].

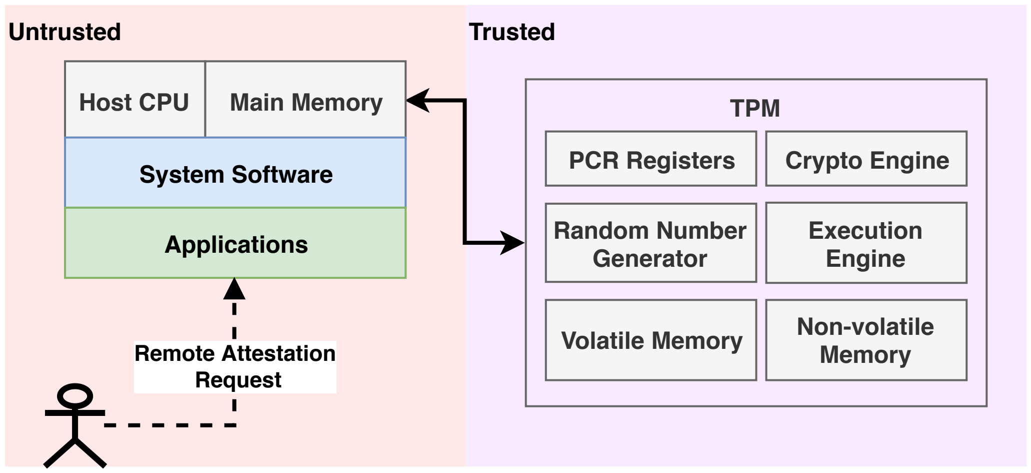


Figure 5.1: The trusted components of a TPM include the PCR registers, crypto engine, and random number generator. Other hardware components, system software, and applications are considered untrusted.

TPMs are secure elements which are typically dedicated physical chips with Common Criteria certification at EAL 4 and higher, and thus provide a very high level of security assurance for the services they offer [67]. As shown in Figure 5.1, the TPM, including components like cryptographic engines, forms the root of trust. On a commodity computer, the host CPU connects to the TPM via a standard communication interface [332]. For trusted execution of cryptographic protocols, applications can request that the OS interact with the TPM device and use various cryptographic engines that support hash functions, encryption, and digital signatures. The TPM also contains non-volatile memory for secure storage of cryptographic parameters and configurations. For instance, a Virtual Private Network (VPN) application can use the TPM to securely store authentication keys and perform authentication without direct access to the private key. TPM also supports remote attestation, in which the TPM will generate a signature using an attestation key derived from the device endorsement key. The manufacturer directly programs the endorsement key into the TPM during manufacturing. Later on, remote parties can use the signature and the public attestation key to attest to the system's integrity. They can use the public endorsement key to verify the integrity of the TPM itself.

5.1.1 TPM Deployment

TPMs have initially been designed as separate hardware modules, but new demands have resulted in software-based implementations. The physical separation of the TPM from the CPU is an asset for protection against system-level adversaries [26]. However, its lightweight design and low-bandwidth bus connection prevent the TPM from being used as a secure cryptographic co-processor for high-throughput applications. TEE technologies such as ARM TrustZone [22] are a more recent approach to bringing trusted execution right into the CPU, at minimal performance loss.

Firmware TPMs (fTPM) can run entirely in software within a TEE like ARM Trustzone [276]. In a cloud environment, the hypervisor executes a software-virtualized TPM device within its trust boundary [122, 239, 271]. In this case, user applications still benefit from the defense against attacks on the guest OS. Virtual TPMs may or may not rely on physically present TPM hardware. Intel Platform Trust Technology (PTT), introduced in Haswell CPUs, is based on fTPM and follows a hybrid hardware/software approach to implement the TPM 2.0 standard. By enabling Intel PTT, computer manufacturers do not need to deploy dedicated TPM hardware.

Intel firmware-based TPM. The Intel management engine (ME) provides hardware support for various technologies such as Intel Active Management Technology (AMT), Intel SGX Enhanced Privacy ID (EPID) provisioning and attestation, and platform trust technology (PTT) [358]. Intel ME is based on an embedded co-processor integrated into all Intel chipsets. This co-processor runs modular firmware on a tiny microcontroller. Since the Skylake generation, Intel has used the MINIX3 OS running on a 32-bit Quark x86 microcontroller, which has an operating frequency of 32 MHz [178]. In particular, these firmware modules and the cryptographic module provide commonly used functions for various services. Previous reverse-engineering efforts have uncovered some of the secrets of the Intel ME implementation [316]. They show that attackers can abuse classical software flaws and vulnerabilities related to the JTAG to compromise Intel ME [99, 100, 101].

Intel PTT, which is essentially a firmware-based TPM, has been implemented as a module that runs on top of the Intel Management Engine (ME). Intel PTT executes on a general-purpose microcontroller, but since it runs independently from the host CPU components, it resembles a more secure hybrid approach than the original Intel fTPM [276], which executes on a TEE on the same core. The exact implementation of the cryptographic functions shared by Intel PTT, EPID, and other cryptographically relevant services is not publicly available.

5.1.2 Vulnreabilities and Shortcomings

The traditional communication interface between dedicated TPM hardware and the CPU is the Low Pin Count (LPC) bus, which is vulnerable to passive eavesdropping [214]. Additionally, researchers have managed to compromise the PCRs based on short-circuiting the LPC pins [196, 319]. However, for cryptographic-coprocessors on the system-on-chip like the Intel fTPM, these attacks are naturally mitigated.

Additionally, researchers have demonstrated other engineering flaws due to the BIOS and bootloader [57, 196], and attacks, exploiting vulnerabilities related to the TPM power management [142]. Nemec et al. developed the "Return of Coppersmith's Attack" (ROCA), which demonstrated passive RSA key recovery from the public key resulting from the particular structure of primes generated on TPM devices manufactured by Infineon [257]. The remote timing attacks that we demonstrate are orthogonal to the key generation issues responsible for ROCA. Our work also focuses on the black-box firmware executing the more complex cryptographic operations. Spark. et al. [319] warn the danger of timing attacks on TPMs, but to the best of our knowledge, nobody showed such attacks on TPMs, as we demonstrate a class of remote timing attack against TPM devices.

As defined by the Trusted Computed Group (TCG), the TPM attempts to mitigate the threat of physical attacks and side channels through a rigorous and lengthy evaluation and certification process. Most physical TPM chips have been certified according to Common Criteria, which involves evaluation through accredited testing labs. Testing labs conduct security evaluations according to protection profiles. For TPM, a specific TCG protection profile exists, which requires the TPM to be secure against side-channel attacks, including timing attacks [341, p. 23].

However, TPMs have previously suffered from vulnerabilities due to weak key generation even on certified devices [257]. However, the industry's widespread belief is that cryptographic algorithms' execution is secure even against system adversaries. Indeed, TPM devices must provide a more reliable root of trust than the OS by keeping cryptographic keys secure. Contrary to this belief, we show in Section 5.2 that these implementations can be vulnerable to remote timing attacks. These attacks reveal cryptographic keys and render modern applications using the TPM less secure than without the TPM.

5.2 Remote Timing Attacks on TPM

Side-channel attacks are a potential attack vector for secure elements like TPMs. These attacks exploit the unregulated physical behavior of a computing device to leak secrets. Processing cryptographic keys may expose secret-dependent signal patterns through physical phenomena. such as power consumption, electromagnetic emanations, or timing behavior [54, 229, 275]. A passive adversary who observes such signals can reconstruct cryptographic keys and break the confidentiality and authenticity of a computing system [85, 236].

Timing attacks. Kocher showed that the secret-dependent timing behavior of cryptographic implementations leaks secret keys [209]. Since then, constant-time operation, or at least secret-independent execution time, has become a standard requirement for cryptographic implementations. For example, the Common Criteria evaluation of cryptographic modules, typical for standalone TPMs, includes testing timing leakage. Brumley et al. showed that remote timing attacks could be feasible across networks by mounting an attack against RSA decryption executed by OpenSSL [54]. Similarly, the OpenSSL ECDSA implementation was vulnerable to remote timing attacks [53]. The latter also showed how lattice-based techniques are powerful tools to recover private keys based on nonce information. Researchers have also demonstrated timing attacks against the implementation of cryptographic protocols. For example, both the Lucky 13 attack [106] and Bleichenbacher's RSA padding oracle attack [237] exploit remote timing. However, the practicality of such attacks against commodity computers has been questioned due to noise and low timing resolution [376]. In comparison, we show that such timing attacks have a more significant impact on TPMs, because of the high-resolution timing information and their specific threat model of a system-level attacker.

Contribution. In this section, we perform a black-box timing analysis of TPM 2.0 devices deployed on commodity computers. Our analysis reveals that elliptic curve signature operations on TPMs from various manufacturers are vulnerable to timing leakage, leading to the private signing key's recovery. In particular, we discovered timing leakage on an Intel firmware-based TPM as well as a hardware TPM. As part of this study, We release an analysis tool that can accurately measure TPM operations' execution time on commodity computers. Our advanced tool supports analysis of command response buffer (CRB) and TPM Interface Specification (TIS) communication interfaces. Later in Chapter 6, we show how this information allows an attacker to apply lattice techniques to recover 256-bit private keys for ECDSA and ECSchnorr signatures. The TPM 2.0 standard supports these elliptic curve primitives, ECDSA and ECSchnorr signature schemes, and the pairing-friendly BN-256 curve used by the ECDAA signature scheme that we found all of them to be vulnerable. We show that this leakage is significant enough to be exploited remotely by a network adversary.

Our study shows that these vulnerabilities exist in devices that have been validated based on FIPS 140-2 Level 2 and Common Criteria (CC) EAL 4+, the highest internationally accepted assurance level in CC, in a protection profile that explicitly includes timing side channels. Even certified devices that claim resistance against attacks require additional scrutiny by the community and industry as we learn more about these attacks. The vulnerabilities we have uncovered emphasize the difficulty of correctly implementing known constant-time techniques and show the importance of evolutionary testing and transparent evaluation of cryptographic implementations.

Experimental setup. We tested Intel fTPM on multiple computers running Intel Management Engine (ME), and we demonstrate key recovery attacks on these machines. We also tested multiple machines manufactured with dedicated TPM hardware, as discussed in Section 5.2.3. All the machines run Ubuntu 16.04 with kernel 4.15.0-43-generic. We used the *tpm2-tools*[1] and *tpm2-tss*[2] software packages and the default TPM kernel device driver to interact with the TPM device. Our analysis tool takes advantage of a custom Linux loadable kernel module (LKM).

5.2.1 Precise Timing Measurement

This section describes our custom timing analysis tool. It shows how a privileged adversary can exploit the OS kernel to perform accurate timing measurement of the TPM and discover and exploit timing vulnerabilities in cryptographic implementations running inside the TPM. We then report the vulnerabilities we discovered related to elliptic curve digital signatures. Later, in Section 6.2, we combine the knowledge of these vulnerabilities with the lattice-based cryptanalysis to demonstrate end-to-end key recovery attacks under various practical threat models[3].

The TPM device runs at a much lower frequency than the host CPU, as it is generally implemented based on a power-constrained platform such as an embedded microcontroller. We can use the CPU's cycle count on the Intel CPU as a high-precision time reference to measure an operation's execution time inside the TPM device. To perform this measurement on the host CPU entirely from software while minimizing noise, we need to make sure that we can read the CPU's cycle count right before the TPM device starts executing a security-critical function and after its completion.

The Linux kernel supports device drivers to interact with the TPM that support various common communication standards. We examined the TPM kernel stack and different TPM 2.0 devices on commodity computers. Our observations suggest that Intel fTPM uses the command response buffer (CRB) [333] and dedicated hardware TPM devices use the TPM Interface Specification (TIS) [332] to communicate with the host CPU. The Linux TPM device driver implements a push mode of communication with these interfaces, where the OS sends the user's request to the device and checks in a loop

[1]https://github.com/tpm2-software/tpm2-tools `commit c66e4f0`

[2]https://github.com/tpm2-software/tpm2-tss `commit 443455b`

[3]The source code for our timing analysis tool, lattice attack scripts, and a subset of data set are available at github.com/VernamGroup/TPM-Fail.

Field	**Offset**	**Description**
Request	00	Power state transition control
Status	04	Status
Cancel	08	Abort command processing
Start	0c	A command is available for processing
Interrupt Control	10	Reserved
Command Size	18	Size of the Command (CMD) Buffer
Command Address	1c	Physical address of the CMD Buffer
Response Size	24	Size of the Response (RSP) Buffer
Response Address	28	Physical address of the RSP Buffer

Table 5.1: The CRB control area: The CRB interface does not prescribe a specific access pattern to the fields of the Control Area. The `Start` and `Status` fields are used to start a TPM command and check the status of the device, respectively.

whether the command has completed the operation or not. As soon as the completed status is detected, the OS reads the response buffer and returns the user's results. The status check for this operation initially waits for 20 milliseconds to perform another status check, and it doubles the wait time every time the device is pending.

This push model of communication makes the timing measurement of TPM operations from userspace less efficient and prone to noise. To mitigate the noise, we initially develop a kernel driver that installs hooks into the CRB and TIS interfaces to modify the described behavior and measure TPM devices' timing as accurately as possible. Later, we move to more realistic settings, *i.e.*, noisy user-level access without root privileges, and environments where the TPM is accessed remotely over the network.

CRB timing measurement. CRB supports a `control area` structure to interface with the host CPU. The `control area`, as shown in Section 5.2.1, is defined as a memory-mapped IO (MMIO) on the Linux OS in which the TPM drivers communicate with the device by reading from or writing to this data structure. We install a hook on the `crb_send` procedure responsible for sending a TPM command to the device over the CRB interface. By default, the driver sets the `Start` field in the control area after preparing the command size and address of the command buffer to trigger the execution of the command by the device. Later on, the device will clear this bit when it completes the command. Listing 5.1 shows the modification of `crb_send`, in which the `Start` field is checked in a tight loop after trigger. As a result, the `crb_send` will only return upon completion of the command, and cycle counts are measured as close to the device interface as possible.

```
t = rdtsc();
iowrite32(CRB_START_INVOKE, &g_priv->regs_t->ctrl_start);
while((ioread32(&g_priv->regs_t->ctrl_start) & CRB_START_INVOKE) ==
    CRB_START_INVOKE);
tscrequest[requestcnt++] = rdtsc() - t;
```

Listing 5.1: CRB Timing Measurement

TIS timing measurement. Similarly, the TIS driver uses an MMIO region to communicate with the TPM device. The first byte of this mapped region indicates the status of the device. To measure the TPM's accurate timing over TIS, we install a hook on the `tpm_tcg_write_bytes` procedure. In the modified handler (Listing 5.2), we check if the write operation issued by the TIS driver stack is related to the trigger for the command execution, `TPM_STS_GO`. If this is the case, we check the buffer for `TPM_STS_DATA_AVAIL` status, indicating the completion of the command execution, in a tight loop. Similar to CRB, the cycle counts are measured close to the device interface.

```
enum tis_status {TPM_STS_GO = 0x20, TPM_STS_DATA_AVAIL = 0x10, ...};
int tpm_tcg_write_bytes_handler(struct tpm_tis_data* data, u32 addr, u16 len, u8* value){
   ...
if(len == 1 && *value == TPM_STS_GO && TPM_STS(data->locality) == addr)
{
   t = rdtsc();
   iowrite8(*value, phy->iobase + addr);
   while(!(ioread8(phy->iobase + addr) & TPM_STS_DATA_AVAIL));
   tscrequest[requestcnt++] = rdtsc() - t;
} ...
```

Listing 5.2: TIS Timing Measurement

5.2.2 Timing Analysis of ECDSA

We profiled the timing behavior of the ECDSA signature schemes using the NIST-256p curve. This average cycle count for Intel fTPM is different for each configuration due to the CPU's working frequency, but the average execution time is similar in various configurations: As shown in Section 5.2.2, we report the average number of CPU cycles to compute the ECDSA signatures for the platforms mentioned above. For example, we observe the highest cycle count on the Core i7-7700 machine, a desktop CPU with a base frequency of 3.60 GHz. We can calculate the average execution time for ECDSA on Intel fTPM as 4.7×10^8 cycles$/3.6\ GHz = 130ms$. The Intel fTPM device's working frequency is relatively slow, facilitating our observation of timing vulnerabilities on such platforms. As the numbers for the dedicated hardware TPM chips suggest, there is a significant difference in execution time between various manufacturers' implementations. To test the ECDSA signature scheme, we generated a single ECDSA key using the TPM device and then measured the ECDSA signature generation's execution time on the device.

Machine	**CPU**	**Vendor**	**TPM**	**Firmware/Bios**	**ECDSA** (Cycle)	**RSA** (Cycle)
NUC 8i7HNK	Core i7-8705G	Intel	PTT (fTPM)	NUC BIOS 0053	4.1e8	7.0e8
NUC 7i3BNK	Core i3-7100U	Intel	PTT (fTPM)	NUC BIOS 0076	3.2e8	5.4e8
Asus GL502VM	Core i7-6700HQ	Intel	PTT (fTPM)	Latest OEM	3.5e8	5.9e8
Asus K501UW	Core i7 6500U	Intel	PTT (fTPM)	Latest OEM	3.4e8	5.8e8
Dell XPS 8920	Core i7-7700	Intel	PTT (fTPM)	Dell BIOS 1.0.4	4.7e8	8.0e8
Dell Precision 5510	Core i5-6440HQ	Nuvoton	rls NPCT	NTC 1.3.2.8	4.9e8	1.8e9
Lenovo T580	Core i7-8650U	STMicro	ST33TPHF2ESPI	STMicro 73.04	8.7e7	9.2e8
NUC 7i7DNKE	Core i7-8650U	Infineon	SLB 9670	NUC BIOS 0062	1.4e8	5.1e8

Table 5.2: Tested Platforms with Intel fTPM or dedicated TPM device.

The security of ECDSA signatures depends on the randomly chosen nonce. The TPM device must use a robust random number generator to generate this nonce independently and randomly for each signing operation to preserve the security of the ECDSA scheme [260].

Our analysis reveals that Intel fTPM and the dedicated TPM manufactured by STMicroelectronics leak information about the secret nonce in elliptic curve signature schemes, leading to efficient recovery of the private key. We will discuss these results in Section 5.2.3. We also observe non-constant-time behavior by the TPM manufactured by Infineon, which, as discussed shortly, does not appear to expose an exploitable vulnerability. Figure 5.2 shows that the TPM manufactured by Nuvoton exhibits constant-time behavior for ECDSA.

Infineon ECDSA timing behavior. Figure 5.3 shows that the TPM manufactured by Infineon experiences non-constant-time behavior for ECDSA. We performed a similar analysis by observing the correlation of LZBs in the nonce and timing (Figure 5.4), and we did not observe any exploitable bias based on the timings. We also performed other intuitive tests, such as looking at the correlation between the timing behavior and the occurrence of 1s. None of our tests were successful in finding time-dependent bias in the nonce.

RSA timing behavior. Using the methodology described earlier, we also profiled the timing behavior of the RSA signature scheme. In Section 5.2.2, we report the average number of CPU cycles to compute RSA signatures for five configurations that support Intel fTPM and three different configurations with a dedicated TPM chip. For this test,

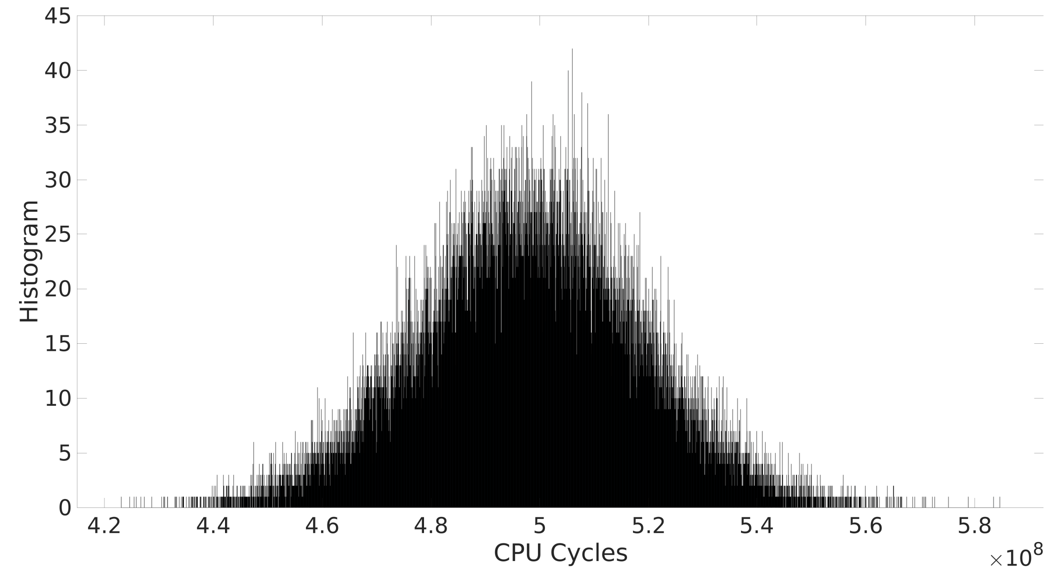


Figure 5.2: Histogram of ECDSA (NIST-256p) signature generation timings a dedicated Nuvoton TPM as measured on a Core i5-6440HQ machine for 40,000 observations.

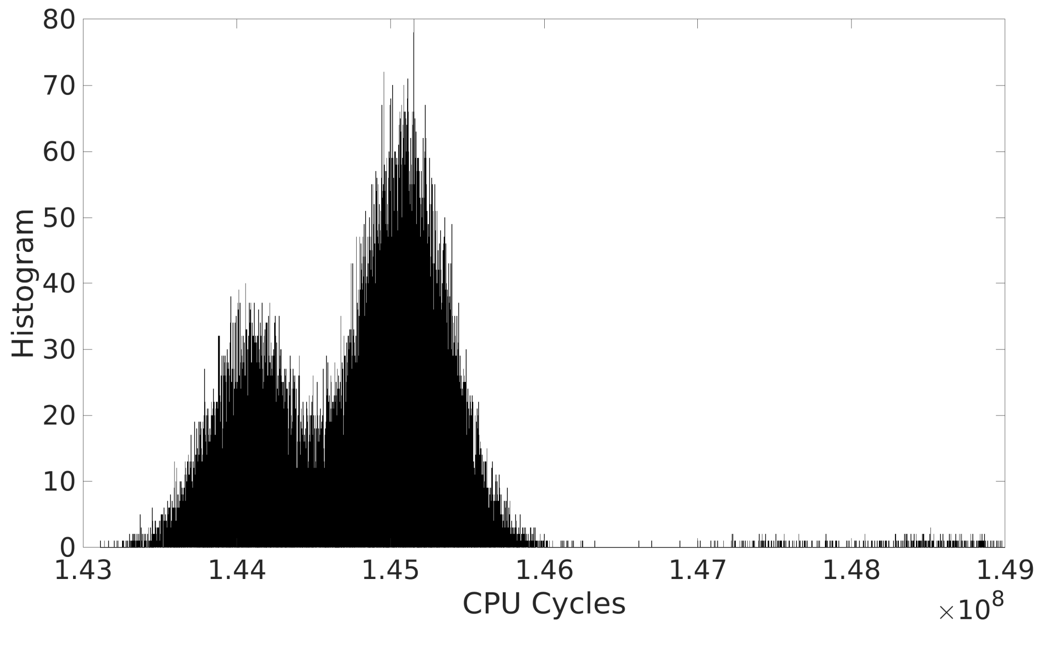


Figure 5.3: Histogram of ECDSA (NIST-256p) signature generation timings a dedicated Infineon TPM as measured on a Core i7-8650U machine for 40,000 observations.

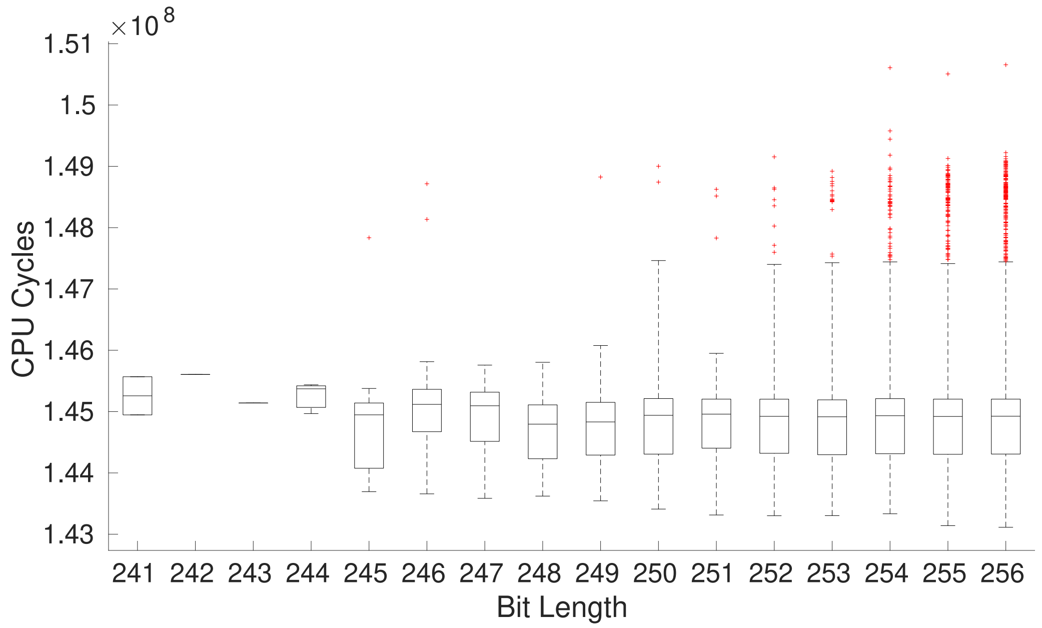


Figure 5.4: Box plot of ECDSA (NIST-256p) signature generation timings a dedicated Infineon TPM as measured on a Core i7-8650U machine for 40,000 observations.

we generated 40,000 valid 2048-bit RSA keys, programmed the TPM with these keys one at a time, and measured timings for RSA signing operations on the TPM.

The timing distributions for the dedicated TPM devices manufactured by Infineon and STMicroelectronics are relatively uniform, as shown in Figure 5.5 and Figure 5.6. In contrast, the distributions in Figure 5.7 and Figure 5.8 show that RSA signature generation is not constant time on Intel fTPM and the dedicated Nuvoton TPM; instead, it has a logarithmic timing distribution that depends on the key bits.

We have previously observed this type of key-dependent timing behavior for the RSA implementation of Intel's IPP Cryptography library [368]. Intel IPP implements RSA based on the Chinese Remainder Theorem (CRT) [90]. The timing variation is due to the modular inversion operation's use of the recursive Extended Euclidean Algorithm (EEA)[4]. After it computes the signature's CRT components, the EEA is employed to calculate the modular inverses needed to reconstruct the final signature. EEA performs modular reductions using division and recurses according to the Euclidean algorithm until the remainder is zero. In this case, the observed timing behavior leaks the number of divisions. Although we observe key-dependent leakage, the EEA algorithm operates serially, and we may only recover a few initial bits of independent RSA keys. This leakage does not seem

[4]During disclosure Intel also confirmed that a version of the Intel IPP Cryptography library was running in Intel fTPM.

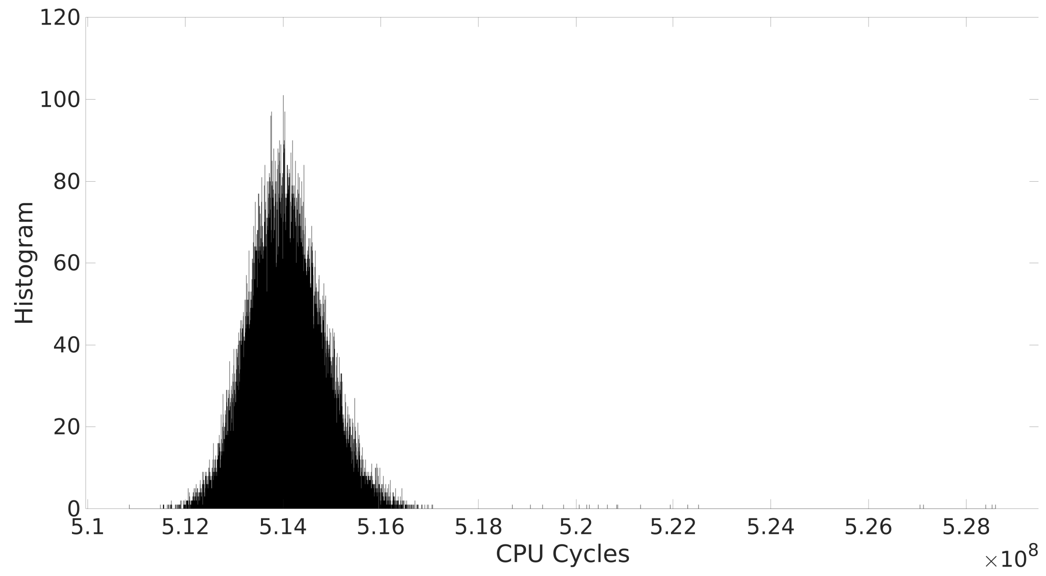


Figure 5.5: Histogram of RSA-2048 signature generation timings on a dedicated STMicroelectronics TPM as measured on a Core i7-8650U machine for 40,000 observations.

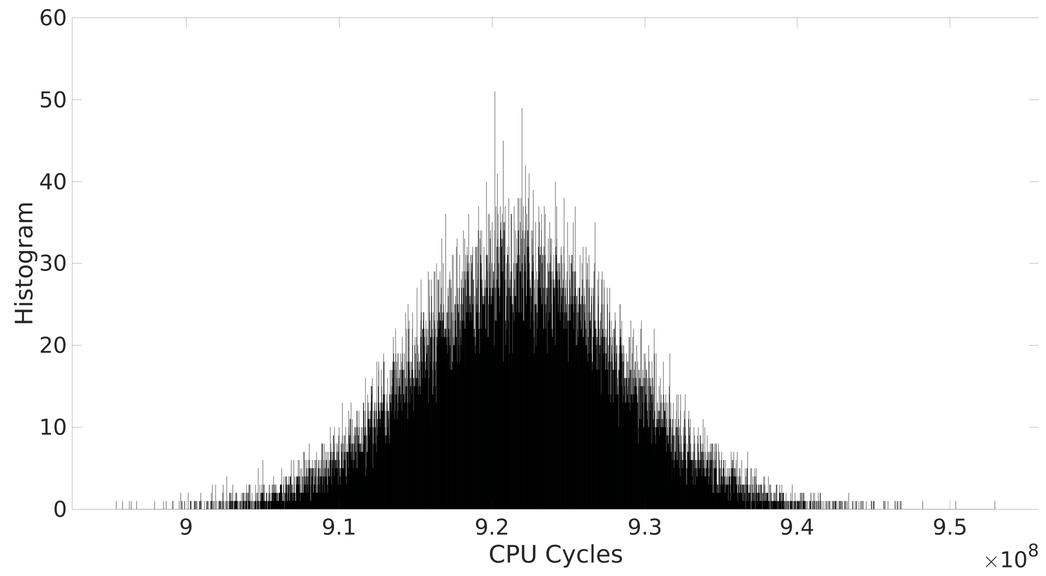


Figure 5.6: Histogram of RSA-2048 signature generation timings on a dedicated Infineon TPM as measured on a Core i7-8650U machine for 40,000 observations.

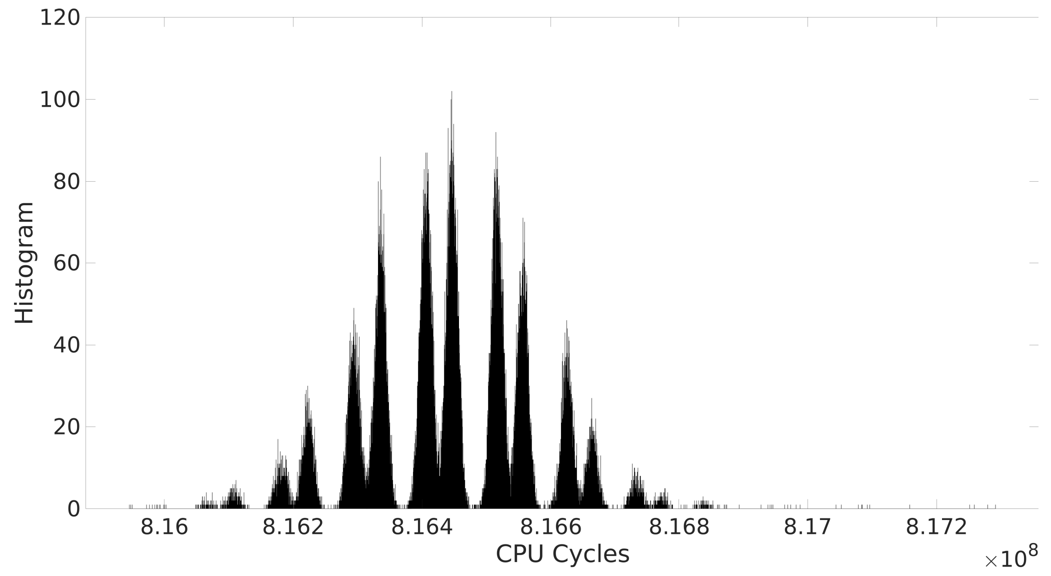


Figure 5.7: Histogram of RSA-2048 signature generation timings on Intel fTPM as measured on a Core i7-7700 machine for 40,000 observations.

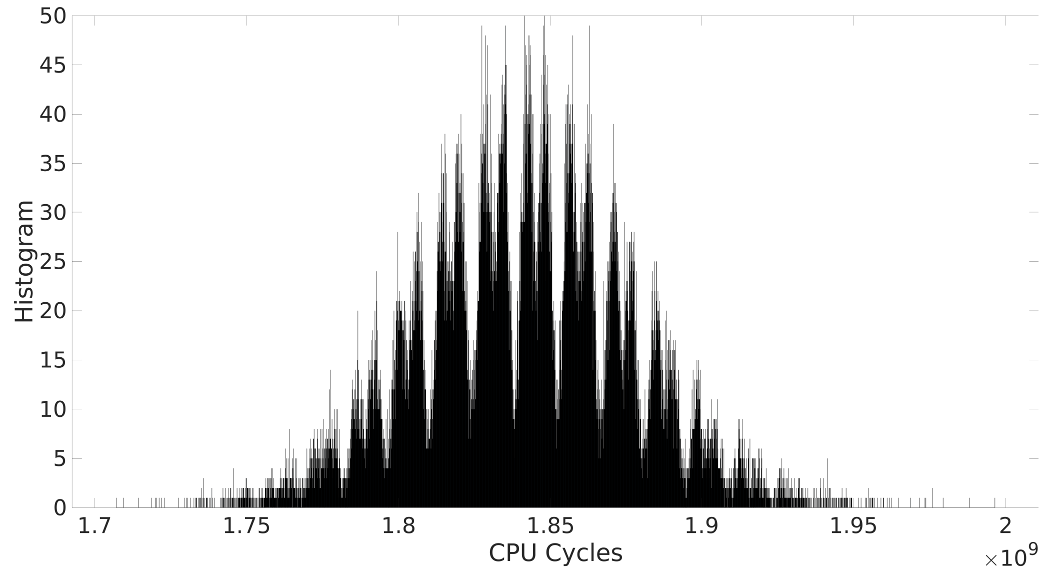


Figure 5.8: Histogram of RSA-2048 signature generation timings on a dedicated Nuvoton TPM as measured on a Core i5-6440HQ machine for 40,000 observations.

to be sufficient for recovery of the full RSA keys using lattice-based or similar methods discussed in Section 6.2, which require a larger proportion of known bits of the secret key for full RSA key recovery [76].

5.2.3 Discovered Vulnerabilities

STMicroelectronics ECDSA scalar multiplication. Figure 5.9 shows an uneven distribution for the STMicroelectronics TPM where there is more leading zero bits (LZBs) left side of the distribution. We used the private key d to compute each nonce k_i for each profiled signature (r_i, s_i) by computing $k_i = s_i^{-1}(H(m) + dr_i) \bmod n$. Figure 5.10 shows a linear correlation between the execution time and the nonce's bit length. This observation shows that the cycle count for each additional zero bit differs by an average of 2×10^5 cycles. This leakage pattern suggests a bit-by-bit scalar point multiplication implementation that skips the computation for the nonce's most significant zero bits. As a result, nonces with more leading zero bits contribute to faster computation.

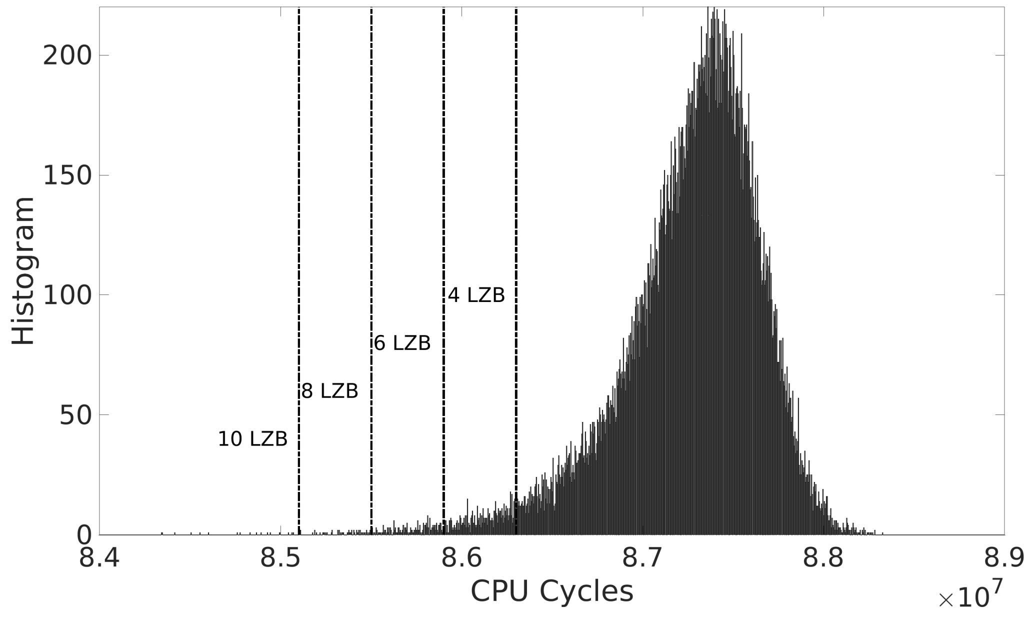


Figure 5.9: Histogram of ECDSA (NIST-256p) signature generation timings on the STMicroelectronics TPM as measured on a Core i7-8650U machine for 40,000 observations.

Intel fTPM ECDSA scalar multiplication. Figure 5.11 shows three distinguishable peaks centered around 4.70, 4.74, and 4.78. Scalar multiplication algorithms to compute $r = (kQ)x$ are commonly implemented using a fixed-window algorithm that iterates

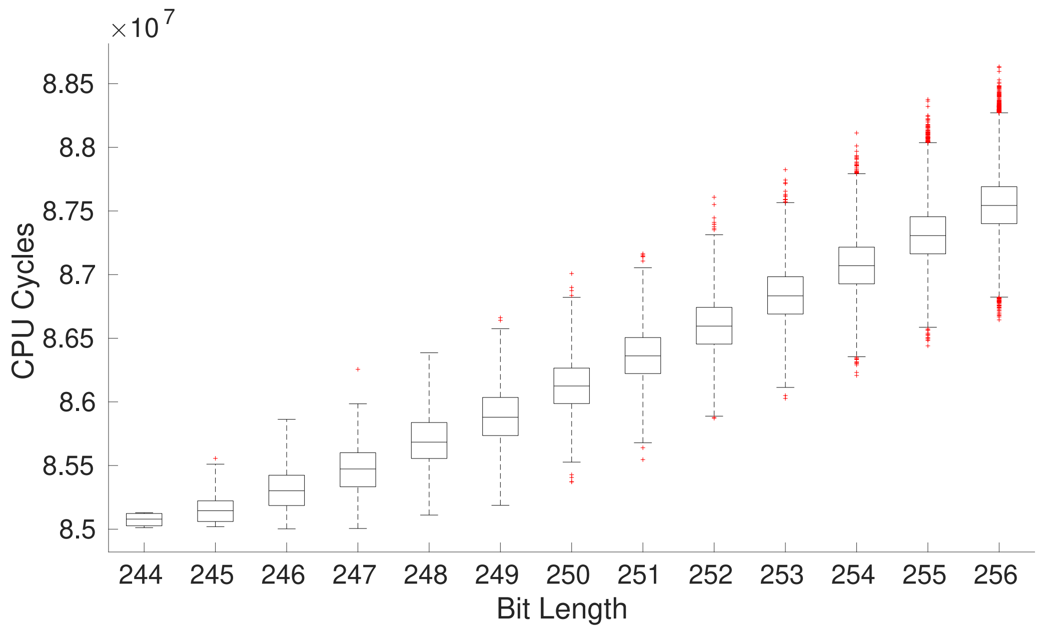


Figure 5.10: Box plot of ECDSA (NIST-256p) signature generation timings by the bit length of the nonce. We observe a clear linear relationship between the two for the STMicroelectronics TPM. Each box plot indicates the median and quartiles of the timing distribution.

window by window over the nonce's bits to calculate the product kQ of the scalar k and point Q. In some implementations, the most significant window (MSW) starts at the first non-zero window of most significant bits of the scalar, which may leak the number of leading zero bits of the scalar [82]. Concerning the observed leakage behavior (Figure 5.11), we expect that:

- The slowest signatures clustered in the rightmost peak represent those with full length k, or in other words, those that have a non-zero most significant window.
- The faster signatures clustered in the second peak may represent signatures computed using nonces k_i that have a full zero MSW but a non-zero second MSW.
- The faster signatures clustered in the third peak may represent signatures computed using nonces k_i that have two full zero MSWs.
- Nonces with three full MSWs of zero bits generated the fastest signatures on the left peak.

The peaks' relative sizes suggest that the implementation we tested uses a 4-bit fixed window (Figure 5.12). This result demonstrates evident leakage of the nonce's length,

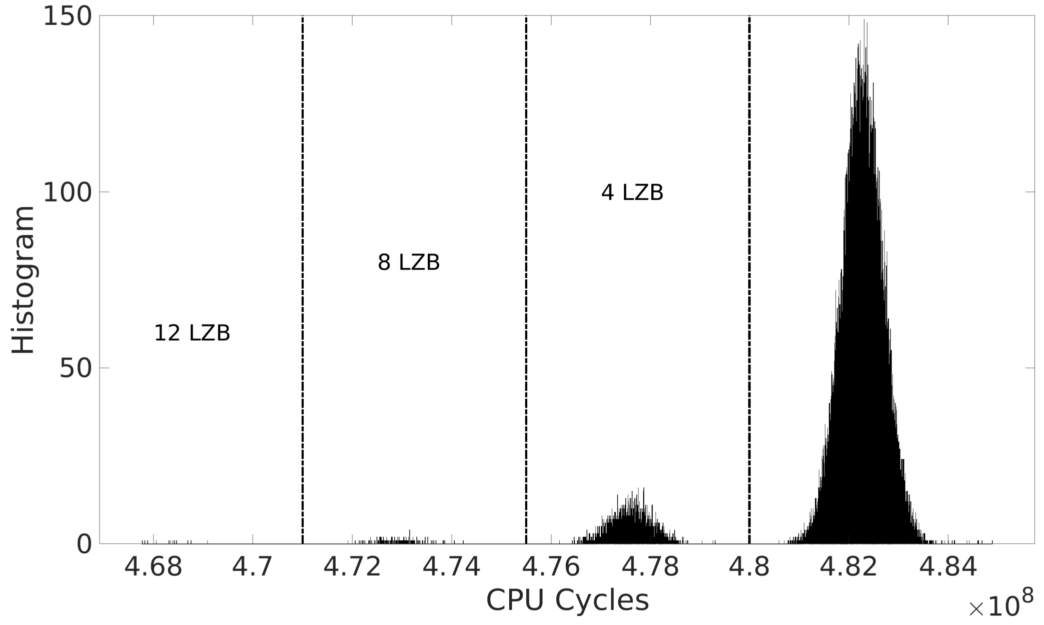


Figure 5.11: Histogram of ECDSA (NIST-256p) signature generation timings on Intel fTPM as measured on a Core i7-7700 machine for 40K observations.

Algorithm 2 Fixed Window Scalar Multiplication

```
1: $T \leftarrow (O, P, 2P, \ldots, (2w-1)P)$
2: procedure MulPoint(window size $w$, scalar $k$ represented as $(k_{m-1}, \ldots, k_0)_{2w}$)
3:     $R \leftarrow T[(k)_{2w}[m-1]]$
4:     for $i \leftarrow m-2$ to $0$ do
5:         for $j \leftarrow 1$ to $w$ do
6:             $R \leftarrow 2R$
7:     return $R$
```

which we can easily exploit using a lattice attack. To summarize, Algorithm Algorithm 2 matches the observed timing behavior of the scalar multiplication inside the Intel fTPM. This observation also aligns with previous vulnerabilities [368] which affected earlier versions of Intel IPP cryptography library [176].

Intel fTPM ECSchnorr scalar multiplication. The ECSchnorr algorithm also uses a secret nonce and scalar multiplication as the first signature generation operation. We performed a similar experiment as above, this time using the `tpm2_quote` command of the TPM 2.0 device. `tpm2_quote` generates a signature using the configured key, but the signature is computed over the PCR registers rather than an arbitrary message. The timing observations suggest that ECschnorr executes about 1.4 times faster than ECDSA,

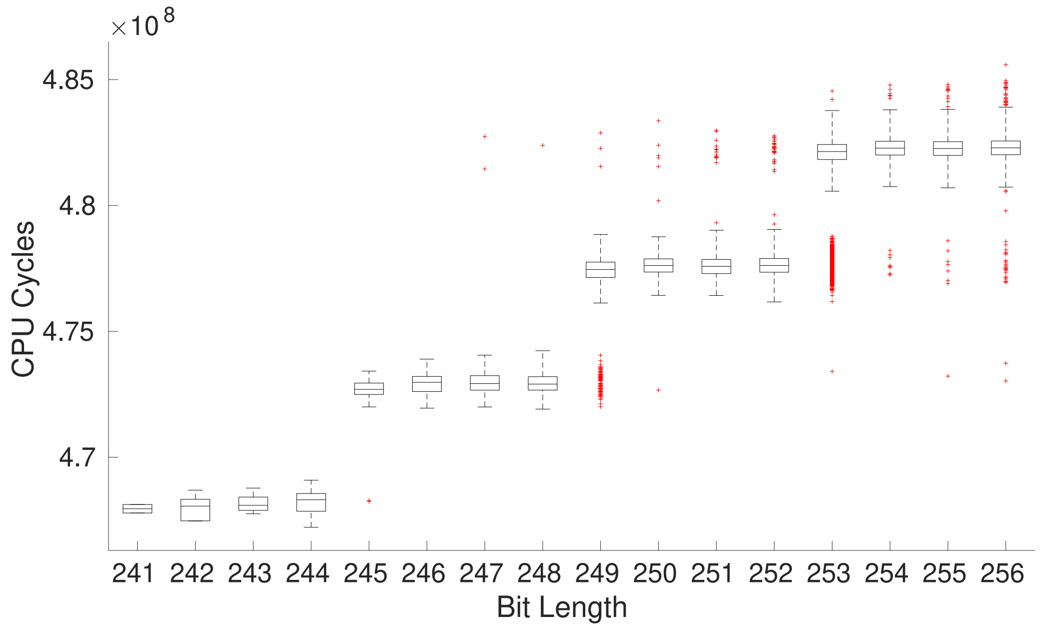


Figure 5.12: Box plot of ECDSA (NIST-256p) signature generation timings depending on the nonce bit length shows a clear step-wise relationship between the execution time and the bit length of the nonce for Intel fTPM.

which implies an independent implementation, but one that is still vulnerable to the same class of timing leakage. The vendor acknowledged this as a separate vulnerability during the bug bounty program. However, they have only assigned CVE-2019-11090 for all issues.

Intel fTPM BN-256 curve scalar multiplication. As mentioned earlier, TPM 2.0 also supports the pairing friendly BN-256 curve, which is used as part of the ECDAA signature scheme. To simplify our experiment and verify that ECDAA is also vulnerable, we configured ECDSA to operate using the BN-256 curve rather than attacking the ECDAA scheme. The timing observation of ECDSA is almost doubled by using the BN-256 curve. It is also vulnerable, as it leaks the leading zero bits of the secret nonce.

5.3 Summary

Since TPMs act as a root of trust, most physical TPMs have undergone validation through FIPS 140-2, which includes physical protection and the more rigorous certification based on Common Criteria up to levels of EAL 4+. This certification aims to prevent a wide range of attacks, including physical and side-channel attacks against its cryptographic capabilities. However, this is the second time that the CC evaluation process has failed

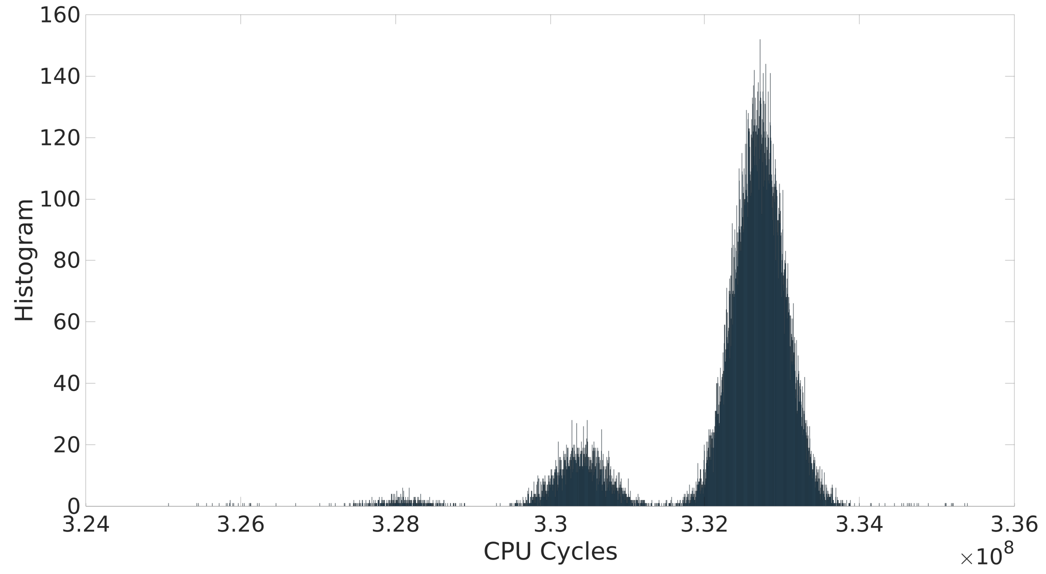


Figure 5.13: Histogram of ECSchnorr (NIST-256p) signature generation times on Intel fTPM as measured on a Core i7-7700 machine for 34,000 observations.

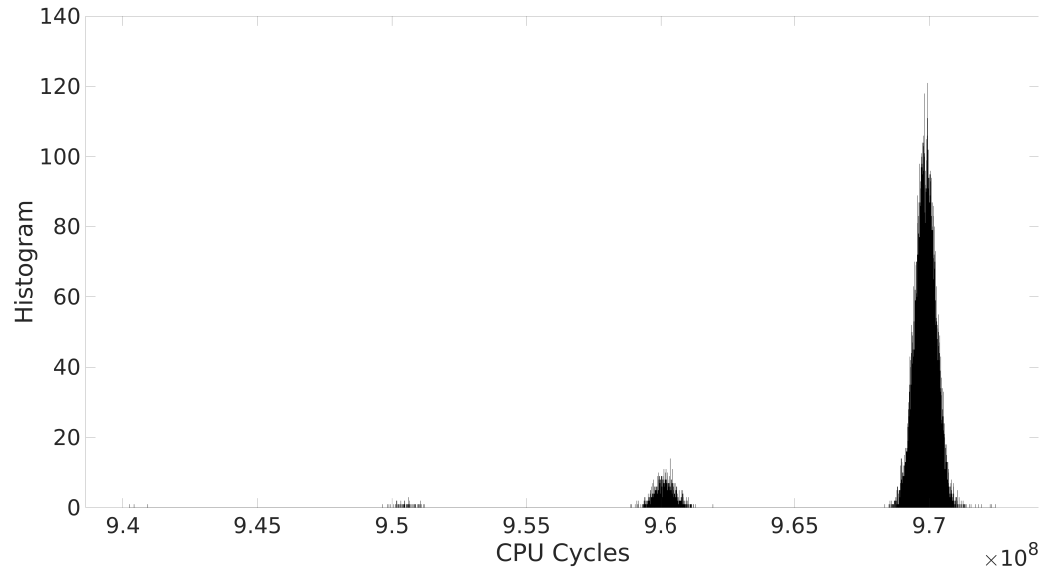


Figure 5.14: Histogram of ECDSA (BN-256) signature generation times on Intel fTPM as measured on a Core i7-7700 machine for 15,000 observations. Using the BN-256 curve approximately doubles the execution time of ECDSA, which makes the multiplication windows even more distinguishable.

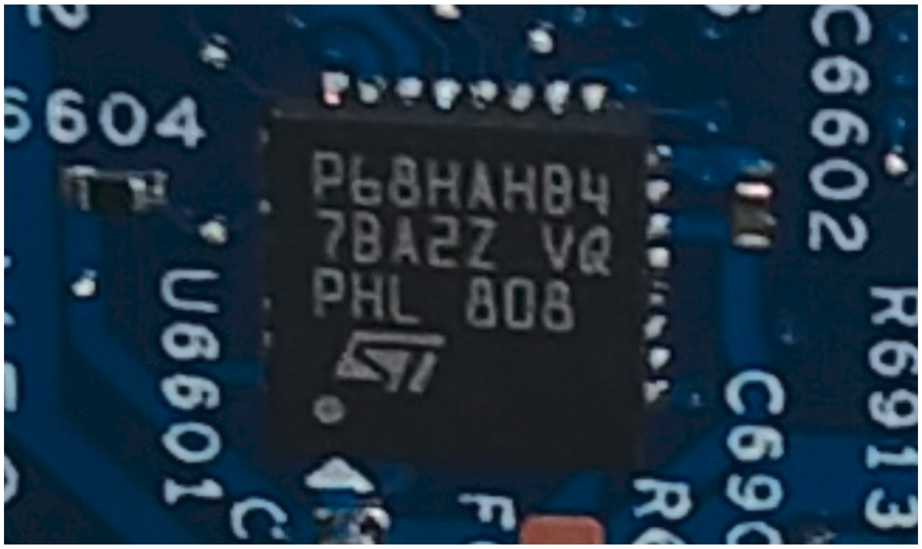


Figure 5.15: The layout of the vulnerable TPM chip on the Lenovo T580 mainboard.

to provide expected security guarantees [257]. These findings underscores the need to reevaluate the CC process. Given the rapid proliferation of side-channel attacks, it would be advisable to switch to a continuously evolving evaluation process. We also note that another potentially vulnerable trusted platform is a Hardware Security Module (HSM). Recent works have already demonstrated that HSMs have more severe vulnerabilities [189]. We expect HSMs to have similar security issues since most have not been certified or tested by an external authority.

The vulnerabilities discovered in this paper apply to a wide range of computing devices. Many PC and laptop manufacturers, including Lenovo, Dell, and HP use the vulnerable **Intel fTPM**. Many new laptop manufacturers prefer using the integrated Intel fTPM rather than adding extra hardware. The Intel fTPM is somewhat comparable to a hardware TPM since it isolates execution in an isolated 32-bit microcontroller. It is also widely used by the Intel IoT platform. Our results on the STMicroelectronics TPM, however, show that even OEMs making a conservative choice and trusting CC-certified hardware TPMs may fall victim to side-channel key recovery attacks. More specifically, we demonstrated vulnerabilities in Intel fTPM and STMicroelectronics TPM devices. We found additional non-constant execution timing leakage in Infineon and Nuvoton TPMs. We will demonstrate end-to-end attacks to recover ECDSA and ECSchnorr keys by collecting signature timing data with and without administrative privileges (§6.2). Further, we will also recover ECDSA keys from an fTPM-based server running StrongSwan VPN over a noisy network as measured by a client. The fact that a remote attack can extract keys from a TPM device certified as secure against side-channel leakage underscores the need to reassess remote attacks on cryptographic implementations, which had been considered a solved problem.

Chapter 6

Microarchitectural Cryptanalysis

This chapter combines the leakage from our proposed timing and attacks with advanced cryptanalysis techniques to demonstrate end-to-end attacks. Section 6.1 exploits the leakage from MemJam in several key stealing attacks against block ciphers based on the S-Box primitive. Section 6.2 is dedicated to lattice-based attacks against ECDSA implementations. We apply lattice-based cryptanalysis to the timing leakage of the TPM-Fail vulnerabilities in several threat models. Ultimately, we show that these attacks are even practical for remote network adversaries. We also show that, in the adversarial OS threat model of SGX, we can bypass the timing mitigation for such vulnerabilities when we combine the lattice-based technique with the CopyCat attack. In Section 6.3, we focus on a different lattice-based attack based on the Coppersmith technique [76] and reconstruct full RSA keys from partial information of RSA key bits gained from Medusa. In the end, in Section 6.4, we derive new algorithms based on the branch-on-prune technique for RSA key extraction. We apply these algorithms to demonstrate end-to-end single-trace attacks against RSA key generation.

6.1 MemJam-Based Correlation Analysis

Secret-dependent cache activities have motivated researchers and practitioners to protect cryptographic implementations against cache attacks [50, 340]. The most straightforward approach for some cryptographic implementations is to minimize the memory footprint of lookup tables. A single 8-Bit S-Box in the advanced encryption standard (AES) rather than T-Tables makes cache attacks on AES inefficient in a noisy environment. The adversary can only distinguish accesses between 4 different cache lines. Combining small tables with cache state normalization, *i.e.*, loading all table entries into cache before each operation defeats cache attacks in asynchronous mode, where the adversary is only able to perform one observation per operation [268].

More advanced side channels such as exploitation of the thread scheduler [133], cache attack on interrupted execution of Intel Software Guard eXtension (SGX) [245], performance degradation [18] and leakage of other microarchitectural resources [5, 8] remind us of the importance of constant-time software implementations. One way to achieve constant-time memory behavior (against a cache-based adversary) is to adopt small tables in combination with accessing all cache lines on each lookup [340]. The overhead would be limited, and the parallelism we can achieve in modern CPUs minimize this overhead. Another constant-time approach adopted by some public cryptographic schemes is interleaving the multipliers in memory known as scatter-gather technique [51].

Contribution. Using MemJam, we demonstrate the first key recovery attacks on constant-time implementations of all symmetric block ciphers supported in the current Intel Integrated Performance Primitives (Intel IPP) cryptographic library: Triple DES, AES, and SM4. These implementations are optimized for both security and speed, and they were a default choice for SGX enclaves. Further, we demonstrate the first intra-cache-line timing attack on SGX by reproducing the AES key recovery results on an enclave that performs encryption using the aforementioned constant-time implementation of AES. The aforementioned constant-time implementation of AES is part of the SGX SDK source code. Our results show that we can use this side channel to efficiently attack memory-dependent cryptographic operations and bypass proposed protections.

AES, SM4, 3-DES, and RC4 are the only available symmetric ciphers as part of Intel's IPP crypto library [176]. Each implementation has optimizations to hinder cache attacks. The 3-DES and the AES implementations feature a constant cache profile and can thus be considered resistant to most microarchitectural attacks, including cache attacks and high-resolution attacks as described in [245]. MemJam can still extract the keys from both implementations due to the intra-cache-line spatial resolution, as depicted in Figure 2.9. We describe the targeted implementations next and the correlation models we use to steal the secret encryption key.

In these cryptanalysis attacks, we assume that the attacker can measure the time of victim encryption. The attacker further knows which cryptographic implementation executes in the victim machine, but she does not need to see the victim's binary or the S-Box tables' offset.

6.1.1 Breaking Pseudo-Constant-Time 3-DES

3-DES. The *3-DES* encryption algorithm [255] is an extension of the DES (Data Encryption Standard) algorithm. While 3-DES was recently deprecated [256], mainly due to its insufficient block size [35] and the resulting attacks for more massive amounts of encrypted data, it is still supported or even required in many applications: The latest EMVCo specification for payment systems permits its usage without further restrictions [98]. Note that EMVCo is an industry consortium managing a payment system standard created by EuroPay, MasterCard, and Visa (resulting in the EMV trademark). Current members include American Express, MasterCard, Visa, and UnionPay [97]. The current TLS 1.2 standard contains 3-DES as a legacy cipher [88].

Given three different 56-bit keys $\mathcal{K}_1, \mathcal{K}_2, \mathcal{K}_3$ and a 64-bit plaintext block M, 3-DES in Encrypt-Decrypt-Encrypt (EDE) mode calculates the cipher text C as

$$CISDEF3\mathrm{DES}_{\mathcal{K}_1,\mathcal{K}_2,\mathcal{K}_3}(M) = \mathrm{DES}_{\mathcal{K}_3}(\mathrm{DES}^{-1}_{\mathcal{K}_2}(\mathrm{DES}_{\mathcal{K}_1}(M))).$$

DES itself is a Feistel network with 16 rounds. First the plaintext M is permuted using an initial permutation $M'ISDEF\mathrm{IP}(M)$ and then divided into two 32-bit blocks $M' = L_0.R_0$. In round $i \in \{0, \ldots, 15\}$ the algorithm then calculates

$$L_{i+1}ISDEFR_i \qquad \text{and} \qquad R_{i+1} = L_i \oplus f(K_i, R_i)$$

for a given round key K_i. The ciphertext C is obtained by applying the inverse of the initial permutation to the last blocks:

$$CISDEF\mathrm{IP}^{-1}(L_{16}.R_{16}).$$

The Feistel function f (Figure 6.1) takes a 48-bit round key K_i and the current right block R_i, and computes its output by doing the following steps:

1. Expand R_i to 48 bits by generating eight 6-bit blocks

   $$B_{i,j}ISDEFR_i[4j - 1 \bmod 32].R_i[4j + 0] \ldots R_i[4j + 3].R_i[4j + 4 \bmod 32]$$

   for $j \in \{0, \ldots, 7\}$.

2. Partition the round key to eight 6-bit blocks $K_i = K_{i,0} \ldots K_{i,7}$ and set the substitution box inputs as

   $$S^{\mathtt{in}}_{i,j}ISDEFB_{i,j} \oplus K_{i,j}$$

   for each $j \in \{0, \ldots, 7\}$.

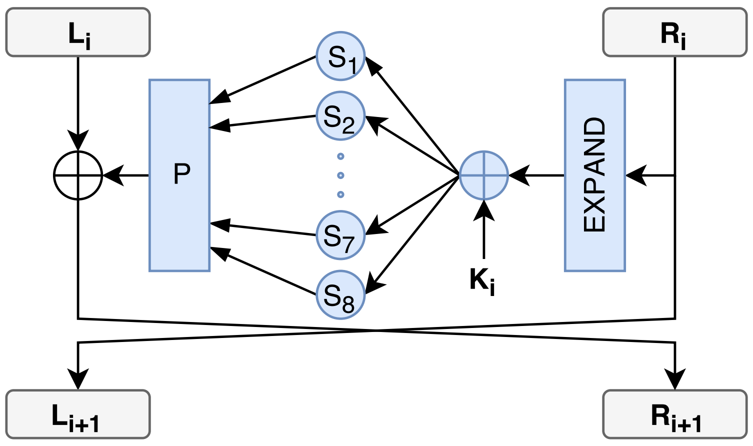


Figure 6.1: Feistel function (blue) in round i of the DES algorithm: First the current right block R_i is expanded to 48 bits and XORed with the round key. Then this value is divided into eight 6-bit blocks, which are substituted by 4-bit blocks using eight different S-boxes. Finally the result is permuted and XORed with the current left block.

3. Use eight S-boxes $\mathrm{S}_0, \ldots \mathrm{S}_7$ to convert the 6-bit inputs into 4-bit outputs:

$$S_{i,j}^{\texttt{out}} ISDEF \mathrm{S}_j(S_{i,j}^{\texttt{in}})$$

for each $j \in \{0, \ldots, 7\}$.

4. Permute the S-Box outputs using a round permutation P to acquire the Feistel function output

$$\mathrm{output} ISDEF \mathrm{P}(S_{i,0}^{\texttt{out}} \ldots S_{i,7}^{\texttt{out}}).$$

The round keys are generated using a schedule consisting of left shifts and permutations [255]; we skip a more in-depth explanation here. Decryption works the same as encryption, except that the decryption operation applies the round keys in reverse order.

Our target, the 3-DES implementation of Intel's Integrated Performance Primitives Crypto library, comes in various flavors where each is optimized for a specific instruction set, but they all have similar cache behavior: The central DES encryption/decryption function `Cipher_DES` first applies the initial permutation, which is implemented as a fixed number of bit operations without any memory accesses. The following 16 rounds are unrolled; each round has exactly $2+16$ memory accesses, where the first two memory accesses load the respective round key. The eight S-box inputs are processed consecutively; for each input **1)** the substitution is performed (by reading from the fixed S-box array), and then **2)** the 4-bit S-Box output is converted into its 32-bit permuted form (using another lookup table). Finally, these permuted outputs are XORed with each other to acquire the result of the Feistel function. Each S-box has $2^6 = 64$ 1-byte entries and therefore fits precisely into one cache line; the same applies to the permutation lookup table, which has $2^4 = 16$ 4-byte entries.

This analysis implies that each cache line is accessed once per round, leading to constant cache behavior that prevents any attacks with cache-line granularity. To obtain data-dependent timing behavior, we used MemJam to induce false dependencies on the first four bytes of the first S-box, slowing down the read accesses to this offset. Since timing behavior slowdown gives us a 4 bytes resolution, we can deduce 4 bits of the respective S-box input, which corresponds to 4 bits of the round key. A single observation consists of the resulting ciphertext C_i, and the number of clock cycles T_i the 3-DES operation takes to execute. Using n of such measurements (with random plaintexts), we can work ourselves into the cipher, starting from the last round.

Single-round attack on 3-DES. Each cipher text C_i consists of blocks $L_{16} = R_{15}$ and R_{16}, where the former directly gives us the eight 6-bit blocks $B_{15,0}, \dots B_{15,7}$. We guess the round key block $K_{15,0}$, and set

- $v[i] ISDEF 1$, if $S^{\texttt{in}}_{15,0} = B_{15,0} \oplus K_{15,0} = \cdot\cdot 0000$
- $v[i] ISDEF 0$, else

for a binary vector $v \in \{0,1\}^n$.

We lose the two least significant bits (written as "$\cdot$") due to the 4-byte resolution of MemJam. Since the IPP implementation reverses each block's bit order and round key, it writes the least significant bits first. Maximizing the correlation

$$\mathrm{corr}(v, T)$$

between the binary vector v and the clock cycle count vector T over all possible round key blocks $K_{15,0}$ then gives us the four key bits $\mathcal{K}_3[2]$, $\mathcal{K}_3[21]$, $\mathcal{K}_3[36]$ and $\mathcal{K}_3[49]$, since the slow runs should be nearly uniformly distributed for wrong guesses.

Multi-round attack on 3-DES. To get the missing 52 key bits, we repeat the attack process in a similar fashion for round 14: The round key block $K_{14,0}$ that we are interested in gives us key bits $\mathcal{K}_3[9]$, $\mathcal{K}_3[28]$, $\mathcal{K}_3[31]$ and $\mathcal{K}_3[43]$, but we also need the last four bits of block $B_{14,0}$; for these, we have to partially calculate $L_{15} = R_{16} \oplus \mathrm{P}(S^{\texttt{out}}_{15,0} \dots S^{\texttt{out}}_{15,7})$, which depends on $K_{15,1}$, $K_{15,4}$, $K_{15,5}$ and $K_{15,7}$, summing up to $4 \cdot 6 = 24$ additional key bits, of which two are already included in the round key $K_{14,0}$.

Repeating the same process for the thirteenth round, in which we need almost all key bits from the fifteenth round to calculate the relevant S-boxes in the fourteenth round, yields another 21 bits of key $\mathcal{K}_3$. We derive the remaining five key bits from round 12. To obtain the remaining keys $\mathcal{K}_1$ and $\mathcal{K}_2$, we repeat the attack using cipher texts decrypted with $\mathcal{K}_3$.

One can also take additional measurements on the other S-boxes to reduce the computational effort, yielding up to 32 key bits in round 15. However, this approach also multiplies the number of measurements, and one still needs to analyze prior rounds to retrieve the missing 24 key bits, although with greatly reduced time complexity. Overall, we see a trade-off between the number of measurements and the computation time spent on the analysis.

3-DES key recovery results on synthetic data. To verify our attack's correctness, we first generated some synthetic data, where the timings remain equal to the number of accesses to the first four bytes of the first S-box. In this noise-free setting, we needed less than 1000 observations to find 19 bits of the 14th round key, with a correlation of 0.201.

3-DES key recovery results using MemJam. The time needed for a successful attack primarily depends on the number of measurements and the number of simultaneously guessed bits. The attacks on round 15 (4 key bits) and 12 (5 key bits) are negligible, but round 14 (26 key bits) needs $2^{26}n$ steps and round 13 (21 key bits) $2^{21}n$ steps; this corresponds to tens of hours of computation time per DES key. While this is significantly less than guessing all 56 bits at once, reducing the number of measurements is still desirable. Figure 6.2 shows the correlations for different measurement counts when guessing 14 key bits in round 14. Experiments showed that 250000-300000 measurements suffice to recover all three keys.

6.1.2 Breaking Pseudo-Constant-Time AES

Advanced Encryption Standard (AES). AES is a cipher based on a substitution permutation network (SPN) with ten rounds supporting 128-bit blocks, and 128/192/256-bit keys [81]. The SubBytes is a security-critical operation, and the straightforward way to implement AES SubBytes operation efficiently in software is to use lookup tables. SubBytes operates on each byte of cipher state, and it maps an 8-bit input to an 8-bit output using a non-linear function. The software can avoid recomputation of this mapping using a precomputed 256-bytes lookup table known as S-Box.

There are efficient implementations using T-Tables that output 32-bit states and combine SubBytes and MixColumns operations. T-Table implementations are highly vulnerable to cache attacks. During AES rounds, a state table is initiated with the plaintext, and it holds the intermediate state of the cipher. Round keys are mixed with

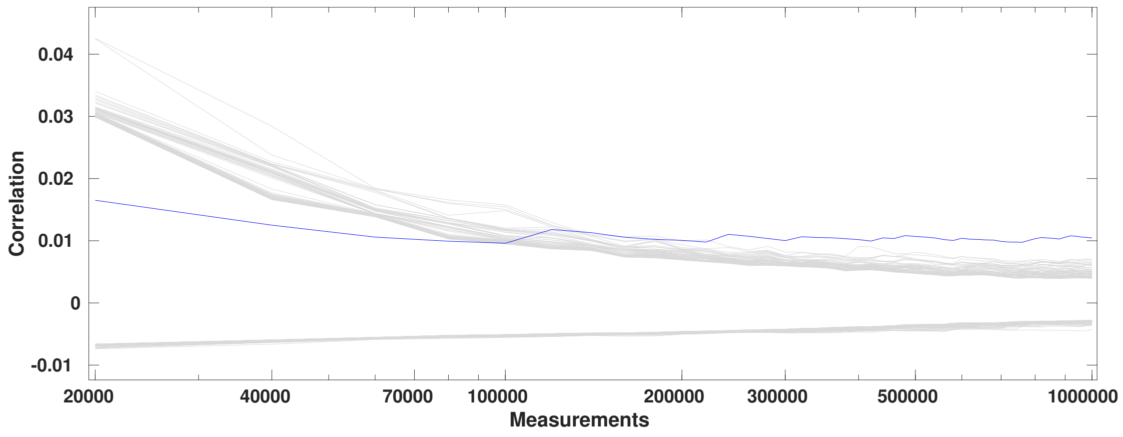


Figure 6.2: The hundred highest and lowest timing correlations when guessing 14 key bits in round 14, depending on the amount of measurements (logarithmic scale). The correct key (blue) becomes distinguishable at around 250000 measurements.

states, which are critical S-Box inputs and the main source of leakage. Hence, even an adversary who can partially determine which entry of the S-Box is accessed can learn some critical information.

Among the efforts to make AES implementations more secure against cache attacks, `Safe2Encrypt_RIJ128` function from Intel IPP cryptographic library is noteworthy. This implementation is the only production-level AES software implementation that features real cache constant-time behavior and does not utilize hardware extensions such as AES-NI or SSSE3 instruction sets. This implementation is also part of the Linux SGX SDK [162], and one can use it for production code if they compile the SDK from scratch, *i.e.*, it does not use prebuilt binaries. We verified the match between the Intel IPP binary and SGX SDK source code implementation through reverse engineering.

This implementation follows a straightforward direction: **(1)** it implements AES using 256 byte S-Box lookups without any optimization such as T-Tables. **(2)** Instead of accessing a single byte of memory on each S-Box lookup, it fetches four values from the same vertical column of 4 different cache lines. It saves them to a local cache aligned buffer. Finally, **(3)** It performs the S-Box replacement by picking the correct S-Box entry from the local buffer. This implementation is depicted in Figure 6.3. This implementation protects AES against any kind of cache attacks, as the attacker sees a constant cache access pattern: The S-Box table only occupies four cache lines, and each SubBytes operation accesses all of them sequentially. This implementation executes in less than 2000 cycles on a recent laptop CPU. This performance is reasonable for many cryptographic applications, and it provides full protection against cache attacks, even if the attacker can interrupt the execution pipeline.

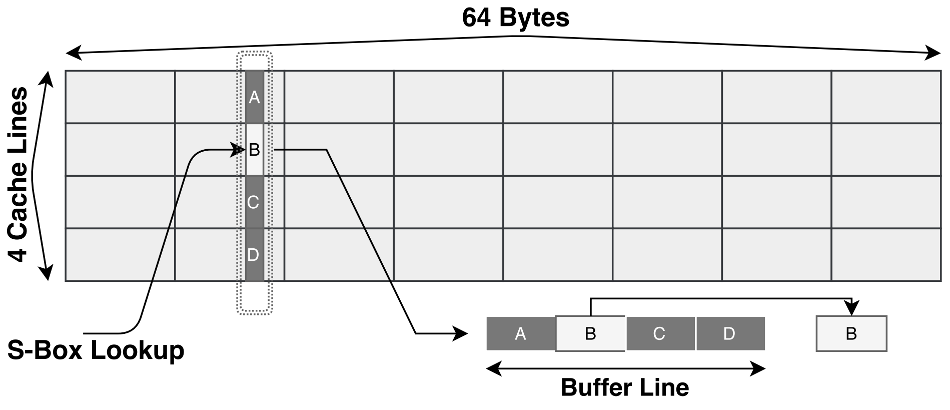


Figure 6.3: Constant-Time table lookup used by Intel IPP: Each lookup preloads 4 values to a cache aligned buffer, thus it accesses all the 4 S-Box cache lines. The actual output will be chosen from the buffer using the high address bits.

Based on the 4-byte granular leakage channel from MemJam, and AES's design, we can create a simple correlation model to attack this implementation. The accessed table index of the last round for a given ciphertext byte c and key byte k is given as $index = S^{-1}(c \oplus k)$. We define matrix $\mathbf{A}$ for the access profile where each row corresponds to a known ciphertext, and each column indicates the number of accesses when $index < 4$. While we assume that the attacker causes slowdowns to the first 4-byte block of S-Box, we define matrix $\mathbf{L}$ for leakage where each row corresponds to a known ciphertext, and each column indicates the victim's encryption time. Then, we define our correlation attack as the correlation between $\mathbf{A}$ and $\mathbf{L}$, in which the higher the number of accesses, the higher the running time. Our results will verify that correlation is high, even though it has dummy accesses to the monitored block. These can be ignored as noise, slightly reducing our maximum achievable correlation.

AES key recovery results on synthetic data. We first verified our correlation model's correctness on synthetic data using a noise-free leakage (generated by PIN [166]). For each of the 16 key bytes using a vector that precisely matches the number of accesses to the targeted block of S-Box for different ciphertexts, all the correct key bytes will have the highest correlation after 32,000 observations with the best and worst correlations of 0.046 and 0.029 respectively.

AES key recovery results using MemJam. Relying on the verification of Synthetic Data, we plugged in the real attack data vector, which consists of pairs of ciphertext and time measured through repeated encryption of unknown data blocks. Results on

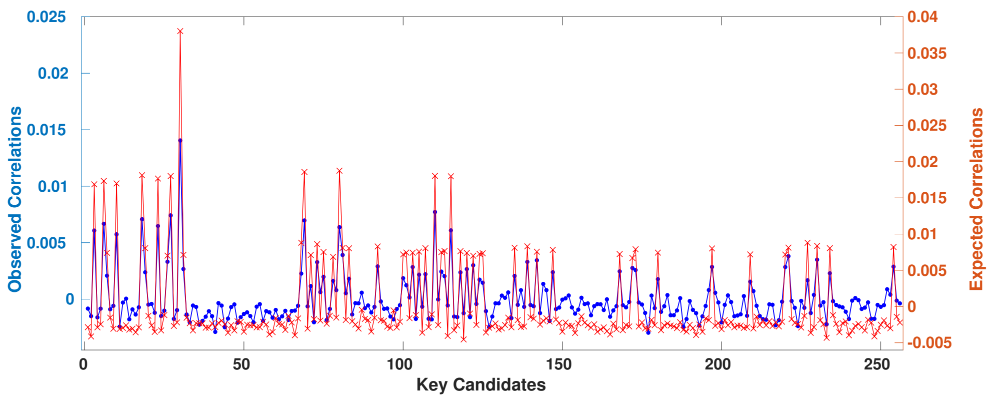


Figure 6.4: Linearity of the number of accesses to the first block and the execution time of AES: The synthetic correlation and MemJam observed correlation show similar behavior with a slight difference due to the added noise.

AES show that we can effectively exploit the timing information and break the so-called constant-time implementation. The AES encryption function's execution takes about 1700 and 2000 cycles without an active thread on the logical CPU pair. The target AES implementation performs 640 memory accesses to the S-Box, including dummy accesses. If the spy thread frequently writes to any address that collides with an S-Box block offset, the time will increase to a range between 2000 and 2300 cycles. The observed variation in this range correlates with the number of accesses to that block.

Figure 6.4 shows the linear correlation between the synthetic data and real attack data for one key byte after 2 million observations. Most of the possible key candidates for a target key byte have a matching peak and hill between the two observations. The highest correlation points in both cases declare the correct key byte (0.038 red, 0.014 blue). The quantitative difference is due to the expected noise in the real measurements.

Figure 6.6 shows the correlation of 4 different key bytes after 2 million observations, with the correct key bytes having the highest correlations. Our repeated experiments with different keys and ciphertexts show that 15 correct key bytes have the highest correlation ranks. Only the byte at index 15 has a high grade but not necessarily the most elevated. Figure 6.7 shows the key ranks over the number of observations. Key byte ranks take values between 1 and 256, where one means that the correct key byte is the most likely one. As it is shown, after only 200,000 observations, the attack reduces the keyspace to a computationally feasible keyspace[120]. After 2 million observations, we recovered all key bytes except one. For most of the key bytes, only tens of thousands of measurements are sufficient to recover the correct key byte (Figure 6.5). The non-optimized implementation of this attack processes 2 million observations in 5 minutes.

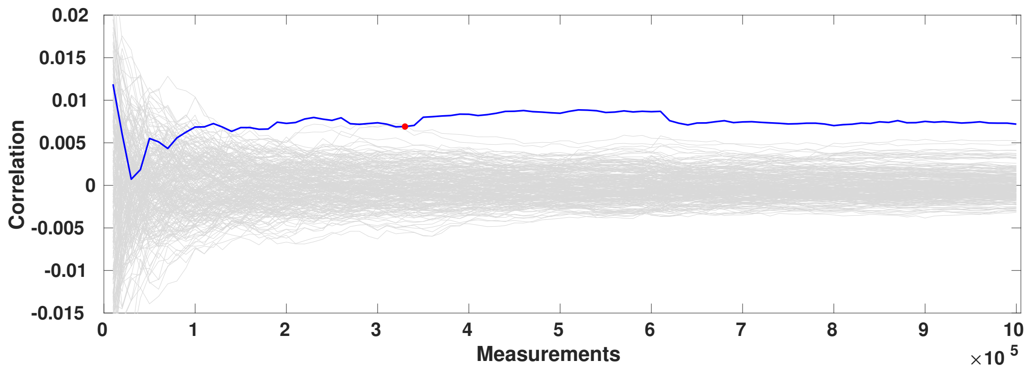


Figure 6.5: The timing correlations for guessing one of the AES key bytes, depending on the amount of measurements. The correct key (blue) becomes distinguishable at around 65000 measurements.

6.1.3 Key Recovery from Cache-Protected SM4

SM4. SM4 (formerly SMS4) is a block cipher standardized by the Chinese government and the standard encryption for Wireless LAN Wired Authentication and Privacy Infrastructure (WAPI) [89]. Several patents investigated by Intel highlight the importance of SM4 [131, 375, 382]. SM4 features an unbalanced Feistel structure and supports 128-bit blocks and keys. SM4 is known to be secure, and no relevant cryptanalytic attacks exist for the cipher. Figure 6.8 shows a schematic of one round of SM4. T1-T4 are 4×32-bit state variables of SM4. Each round mixes the last three state variables and a 32-bit round key, and a non-linear S-Box value will replace each byte of the output. After the non-linear layer, the diffusion layer combines the 32-bit output of S-Boxes x using the linear function L. The production of L is then mixed with the first 32-bit state variable to generate a new random 32-bit state value. The same operation is repeated for 32 rounds, and each time a new 32-bit state is generated as the next round T4 state. The next round treats the current T2, T3, T4 as T1, T2, and T3. The final 16 bytes of the entire state after the last round produce the ciphertext. SM4 Key schedule produces 32×32-bit round keys from a 128-bit key. Since the key schedule is reversible, recovering four repeated round keys provides enough entropy to reproduce the cipher key.

All the SM4 operations except the S-Box lookup have 32-bit word sizes. Hence, SM4 implementation is both efficient and straightforward on modern architectures. We chose the function `cpSMS4_Cipher` from Intel IPP Cryptography library. Our target is the straightforward implementation of the cipher with the addition of S-Box cache state normalization. We recovered this implementation through reverse engineering of Intel IPP binaries. The implementation preloads four values from different cache lines of S-Box before the first round, and it mixes them with some dummy variables, forcing the CPU

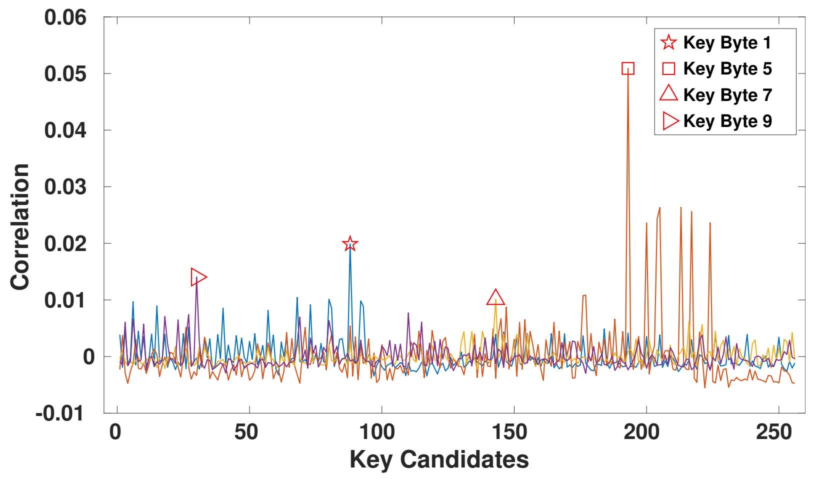


Figure 6.6: Correlations for 4 key bytes using 2 million observations. Correct key byte candidates have the highest correlations.

to fill the relevant cache lines with the S-Box table. This cache prefetching mechanism protects SM4 against asynchronous cache attacks. Our experimental setup runs in about 700 cycles, which informs us that this implementation maintains a high speed while secure against asynchronous attacks. Interrupted attacks that leak intermediate states would not be as simple since the interruption need to happen faster than 700 cycles. We will further discuss the difficulty of correlating any cache-granular information, even if we assume the adversary can interrupt the encryption and perform some intermediate observations.

$$
\begin{aligned}
&x_{32} = c_1 \oplus c_2 \oplus c_3 \oplus k_{32} \\
&d_2 = c_1, d_3 = c_2, d_4 = c_3 \\
&d_1 = L(s(x_{32}^1), s(x_{32}^2), s(x_{32}^3), s(x_{32}^4)) \oplus c_4 \\
&x_{31} = d_1 \oplus d_2 \oplus d_3 \oplus k_{31} \\
&e_2 = d_1, e_3 = d_2, e_4 = d_3 \\
&e_1 = L(s(x_{31}^1), s(x_{31}^2), s(x_{31}^3), s(x_{31}^4)) \oplus d_4 r
\end{aligned}
\qquad
\begin{aligned}
&x_{30} = e_1 \oplus e_2 \oplus e_3 \oplus k_{30} \\
&f_2 = e_1, f_3 = e_2, f_4 = e_3 \\
&f_1 = L(s(x_{30}^1), s(x_{30}^2), s(x_{30}^3), s(x_{30}^4)) \oplus e_4 \\
&x_{29} = f_1 \oplus f_2 \oplus f_3 \oplus k_{29} \\
&g_2 = f_1, g_3 = f_2, g_4 = f_3 \\
&g_1 = L(s(x_{29}^1), s(x_{29}^2), s(x_{29}^3), s(x_{29}^4)) \oplus f_4 \\
&x_{28} = g_1 \oplus g_2 \oplus g_3 \oplus k_{28}
\end{aligned}
\tag{6.1}
$$

Single-round attack on SM4. We define c_1, c_2, c_3, c_4 as the four 32-bit words of a ciphertext and k_r as the secret round key for round r. We recursively follow the cipher structure from the last round with our ciphertext words as inputs and write the last five rounds' relations as Equation 6.1. In each round, x_r^i is the S-Box index, and i is the byte

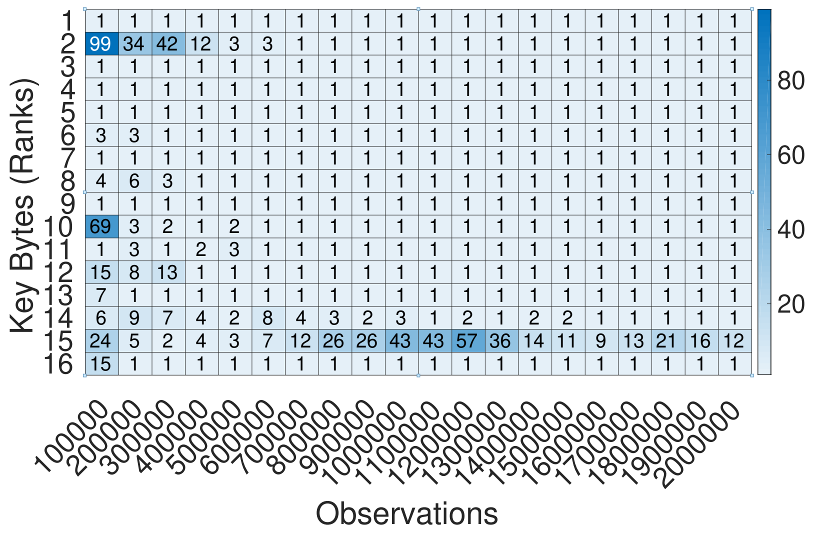


Figure 6.7: The rank for correct key bytes are reduced with more observation. After 2 million observations, 15 out of 16 key bytes are recovered.

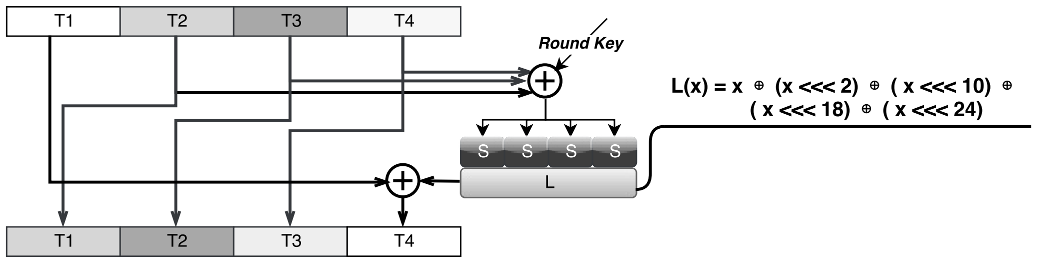

T1
T2
T3
T4
Round Key
L(x) = x ⊕ (x <<< 2) ⊕ (x <<< 10) ⊕ (x <<< 18) ⊕ (x <<< 24)
S
S
S
S
L
T1
T2
T3
T4

Figure 6.8: SM4 Feistel Structure: In each round, the last three words from the state buffer and the round key will be added. An S-Box lookup will replace each byte of the output. The function L performs a linear bit permutation.

offset of the 32-bit word x_r. With a similar approach to the attack on AES, we define matrix $\mathbf{A}$ for the access profile, where each row corresponds to a known ciphertext, and each column indicates the number of accesses when $x_r^i < 4$. Then we define the matrix $\mathbf{L}$ for the observed timing leakage and the correlation between $\mathbf{A}$ and $\mathbf{L}$ similar to the AES attack. In contrast, S-Box indices in the AES attack are defined based on a non-linear inverse S-Box operation of key and ciphertext, which eventually maps all possible key candidates. In SM4, the index x_r^i is defined before any non-linear operation. As a result, an attack capable of distinguishing 4 out of 256 S-Box entries reveals only 6 bits per key byte. In the mentioned relations, performing the attack using this model on x_{32}^i, recovers the six most significant bits of each key byte i for the last round key (Total of 24 out of the 32 bits).

Multi-round attack on SM4. We can use the relationship for round 31 to recover 6-bit key candidates of round 31 and the remaining unknown 8 bits of entropy for round 32. This observation is due to the linear property of function L and the newly created state variables' recursive nature. After the attack on round 32, we only have certainty about 24 bits of the new state variable d_1. Still, this information will be propagated as the input to round 31. The next round of attack for a key byte of round 31 needs more computation to process an 8 bit of unknown key and 8 bit of unknown state (total of 16 bit), but this is computationally feasible. We recover the 8-bit key from round 32 with the highest correlation by attacking the S-Box indices in round 31. We recursively applied this model to each round resulting a correlation attack with the following steps, which gives us enough entropy to recover the key:

1. $x_{32} \rightarrow$ 24 bits of k_{32}.
2. $x_{31} \rightarrow$ 24 bits of k_{31} + 8 bits of k_{32}
3. $x_{30} \rightarrow$ 24 bits of k_{30} + 8 bits of k_{31}
4. $x_{29} \rightarrow$ 24 bits of k_{29} + 8 bits of k_{30}
5. $x_{28} \rightarrow$ 24 bits of k_{28} + 8 bits of k_{29}
6. Recover the key from $k_{32}, k_{31}, k_{30}, k_{29}$

SM4 key recovery results on synthetic data. Our noise-free synthetic data shows that 3000 observations are enough to find all correct 6-bit and 8-bit round key candidates with the highest correlations. Even in an interrupted cache attack or without cache protection, targeting this implementation using cache-granular information would be much more challenging and inefficient due to the lack of intra-cache-line resolution. If we only distinguish the 64-byte cache lines out of a 256-byte S-Box, we only learn 4×2-bit (total of 8 bits) out of 32-bit round keys, and on each round, we need to solve 8 bits + 24 bits of uncertainty. Although solving 32-bit of uncertainty sounds possible for noise-free data, it is computationally much harder in a noisy practical setting. Our intra-cache-line leakage can exploit SM4 efficiently in a known-ciphertext scenario, while the best efficient cache attack on SM4 requires chosen plaintexts [262].

SM4 key recovery results using MemJam. The results on SM4 show even more effective key recovery against this implementation compared to AES. Figure 6.9 shows the correlation rate over measurements for one key byte in the first round of attack, which

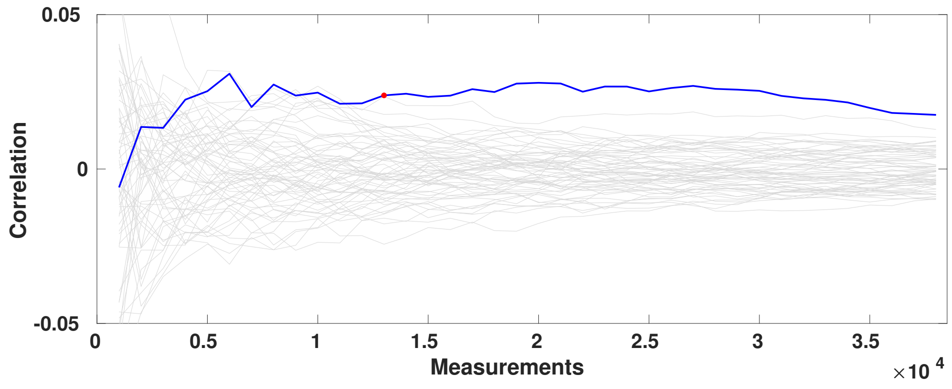


Figure 6.9: The timing correlations for guessing one of the SM4 key bytes in a single round attack, depending on the number of measurements. The correct key (blue) becomes distinguishable at around 13000 measurements.

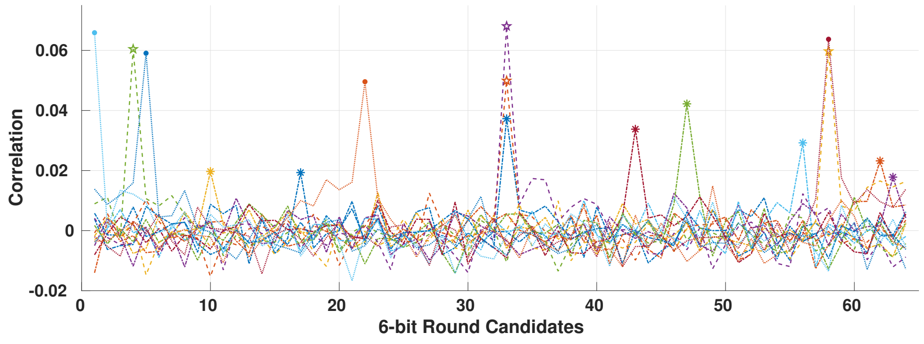


Figure 6.10: Correlations for SM4 6-bit keys of the last 4 32-bit round key recovered through 5 rounds of attack using 40,000 observations.

13000 measurements are sufficient to distinguish the correct 6-bit round key (blue) for this key byte.

Figure 6.10 shows the correlation for 6-bit round keys after five rounds of repeated attack, and we see the correlation for 12-bit key candidates in Figure 6.11. Our attack expects the correctness of key candidates for each round of attack before proceeding to the next round. This property is due to the recursive structure of SM4. In our experiment using real measurement data, we have noticed that 40,000 observations are sufficient to assure correct key candidates with the highest correlations. Our implementation of the attack can recover the correct 6-bit and 8-bit keys, and it takes about 5 minutes to recover the cipher key.

In Figure 6.11, we plotted the accumulated per byte correlations for all 8-bit candidates within each round of attack. During the computation of 6-bit candidates, the 8-bit

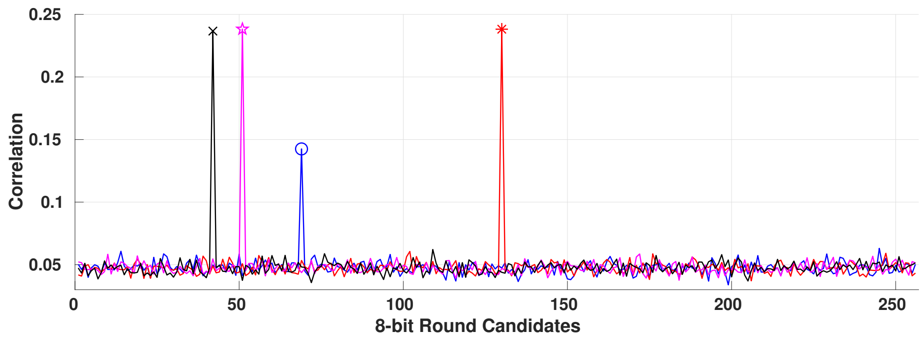


Figure 6.11: The accumulated correlations for SM4 8-bit keys after 5 rounds using 40,000 observations. Each correct candidate has the highest correlation.

candidates relate to 4 different state bytes. This accumulation significantly increases the result, and the correct 8-bit key candidates have a very high aggregated correlation compared to the 6-bit candidates.

6.1.4 MemJam AES Key Recovery Results in SGX

Indeed, Intel ensures constant cache-line accesses for its AES implementation, making it resistant to *all* previously known microarchitectural attacks in SGX. In this section, we verify that MemJam is also applicable to SGX enclaves, as there are no fundamental microarchitectural changes to resist against false memory dependencies. We repeat the key recovery results against Intel's constant-time AES implementation after moving it into an SGX enclave. The results verify the exploitability of intra-cache-line channels against SGX secure enclaves. This attack can be reproduced straightforwardly. The only difference is a slower key recovery due to the increased measurement noise resulting from the enclave context switch.

SGX enclave experimental setup and assumptions. Following the threat model of *CacheZoom* [218, 245], we assume that the system adversary has control over various OS resources. Please note that the goal of SGX is to thwart the threat of such adversaries. The adversary uses its OS-level privileges to decrease the setup noise: We isolate one of the physical cores from the rest of the running tasks and dedicate its logical CPUs to MemJam write conflict thread and the victim enclave. We further disable all the non-maskable interrupts on the target physical core and configure the CPU power and frequency scaling to maintain a constant frequency. We assume that the adversary can measure an enclave interface's execution time that performs encryption, and the enclave interface only returns the ciphertext to the insecure environment.

Both plaintexts and the secret encryption key are generated at runtime using *RDRAND* instruction, and they never leave the secure runtime environment of the SGX enclave. SGX does not allow the *RDTSC* instruction inside an enclave. The attacker uses it right before the call to the enclave interface and again right after the enclave exit. As a result, the entire execution of the enclave interface, including the AES encryption, is measured. As before, an active thread causing read-after-write conflicts to the first four bytes of the AES S-Box is executed on the neighboring virtual CPU of the SGX thread.

Execution of the same AES encryption function as Section 6.1.2 inside an SGX enclave interface takes an average of 14,600 cycles with an active thread causing read-after-write conflicts to the first four bytes of the AES S-Box. The additional overhead is caused by the enclave context switch, which significantly increases the timing channel's noise due to the variable timing behavior. This experiment shows a more practical timing behavior where adversaries cannot time the actual encryption operation, and they have to measure the time for a batch of operations. This observation not only shows that SGX is vulnerable to the MemJam attack, but it also demonstrates that attackMemJam is applicable in a realistic scenario. Figure 6.12 shows the key correlation results using 50 million timed encryptions in SGX, collected in 10 different time frames. We filtered outliers, *i.e.*, measurements with high noise, only considering samples in the range of 2000 cycles of the mean. Among the 50 million samples, 93% pass the filtering, and we only calculated the correlations for the remaining traces. Figure 6.13 shows that we can successfully recover 14 out of 16 key bytes, revealing sufficient information for key recovery after 20 million observations.

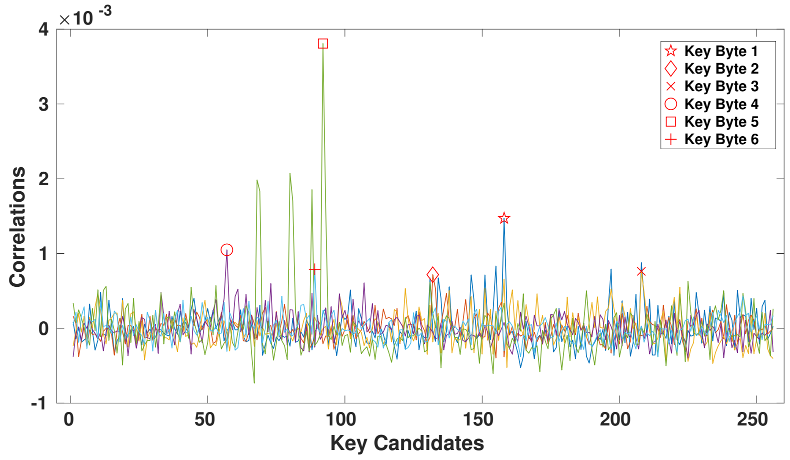
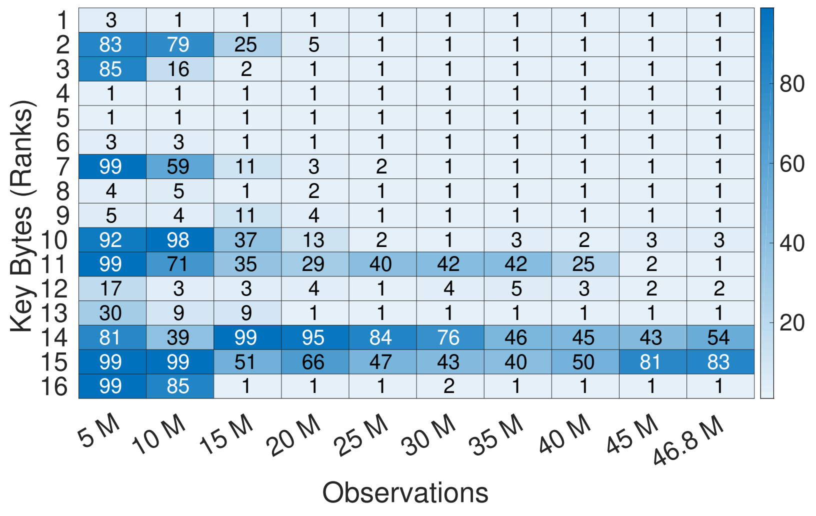


Figure 6.12: Correlations for 6 key bytes using 5 million observations. All of the correct candidates have the highest correlations.



	5 M	10 M	15 M	20 M	25 M	30 M	35 M	40 M	45 M	46.8 M
1	3	1	1	1	1	1	1	1	1	1
2	83	79	25	5	1	1	1	1	1	1
3	85	16	2	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1	1
6	3	3	1	1	1	1	1	1	1	1
7	99	59	11	3	2	1	1	1	1	1
8	4	5	1	2	1	1	1	1	1	1
9	5	4	11	4	1	1	1	1	1	1
10	92	98	37	13	2	1	3	2	3	3
11	99	71	35	29	40	42	42	25	2	1
12	17	3	3	4	1	4	5	3	2	2
13	30	9	9	1	1	1	1	1	1	1
14	81	39	99	95	84	76	46	45	43	54
15	99	99	51	66	47	43	40	50	81	83
16	99	85	1	1	1	2	1	1	1	1

Figure 6.13: The rank for correct key bytes with respect to the number of observations. Using the entire data set, after filtering the outliers, we can recover 14 out of 16 key bytes.

These results show that even cryptographic libraries designed by experts who are fully aware of recent attacks and the target device's leakage behavior may fail at writing non-exploitable code. Modern microarchitectures are so complicated that assumptions such as *constant cache line profiles* result in constant-time implementations that are seemingly impossible to fulfill.

6.1.5 Discussion on MemJam Cryptanalysis

An adversary who performs the MemJam attack also does not need to know about the offset of an S-Box in the binary since she can simply scan the 10-bits address entropy by introducing conflicts to different offsets and measuring the timing of the victim. In such a scenario, we assume that the S-Box table is aligned with the cache line size since an unaligned S-Box in memory is already vulnerable to cache attacks [181, 267]. During the processing of uniformly random input, each S-Box operation of `Safe2Encrypt_RIJ128` accesses the first-word column of the table with a probability of $1/16$. Among 160 S-Box operations, an average of 10 memory accesses to the first S-Box is likely. While an attacker is causing RaW conflicts on increasing offsets, they can locate the S-Box offset as soon as they see a timing behavior. This understanding is essential for obfuscated binaries or scenarios where the offset of the S-Box is unknown.

As shown in Section 6.1.5, all block cipher implementations of IPP have at least one vulnerable variant. In cases where an implementation relies on the AES-NI instruction set (or SSSE3, respectively), the library falls back to the basic version at runtime if

Implementation	Function Name	l9/n0/y8/k0/e9	m7/mx	n8	SGX SDK
DES Constant-Time	Cipher_DES	✓	✓	✓	N/A
AES-NI	Encrypt_RIJ128_AES_NI	✓	×	×	✓ (prebuilt)
AES Bitsliced	SafeEncrypt_RIJ128	✓	×	✓	✓ (prebuilt)
AES Constant-Time	Safe2Encrypt_RIJ128	×	✓	×	✓ (source)
SM4 Bitsliced & AES-NI	cpSMS4_ECB_aesni	✓	×	×	N/A
SM4 Cache Normalized	cpSMS4_Cipher	✓	✓	✓	N/A

Table 6.1: DES, SM4 and AES implementations in all variants of Intel IPP library version 2018 [170]. The linker merges these variants, and each variant optimizes for a different generation of the Intel instruction set [159]. Developers can statically link specific variants with single CPU static linking mode [170].

the instruction set extensions are not available. e The usability of this depends on the compilation and runtime configuration. Developers are allowed to link to a riskier variant [159] statically, and they need to ensure not to use the vulnerable versions during linking. Developers should avoid these ciphers even when the hardware does not support hardware extension, e.g., Core and Nehalem do not support AES-NI; also, AES-NI can be disabled in some BIOS. For 3-DES, IPP gives only one implementation option: the vulnerable one studied in this work. Thus, for applications that demand the use of 3-DES (and there are still many such applications, as discussed in Section 6.1.1), there is no secure alternative available in IPP. This finding highlights that current hardware support for cryptographic primitives is restricted, and if any cipher without explicit hardware support is required, this limitation may endanger the provided security. MemJam is another piece of evidence that modern microarchitectures are too complicated, and constant-time implementations cannot only be trusted, as assumptions about the underlying system often turn out to be wrong.

6.2 Lattice Attacks on ECDSA

This section provides an overview of the digital signature algorithm (DSA) and its variant, elliptic-curve DSA (ECDSA). Then we discuss the hidden number problem and its application in the side-channel analysis of DSA and ECDSA. We demonstrate microarchitectural cryptanalysis of several deployed ECDSA implementations, deployed by TPMs and cryptographic libraries, after this background information.

6.2.1 Digital Signature Algorithms

Authentication and remote attestation for secure elements and TEEs extensively use such signature schemes [174]. Moreover, TEEs like Intel SGX can promise trusted execution of these algorithms for a wide range of applications such as trusted key management [109] and private contact discovery [314]. We provide an overview of Schnorr-type signing [306].

DSA. In the DSA [112], the public parameters are a prime p, another prime divisor n of $p-1$, and the group generator g. The private key x is chosen randomly such that $1 < x < n-1$, and the public key is $y = g^x (\text{mod} p)$. To sign a message hash h:

1. Choose a random secret k such that $1 < k < n-1$,
2. Compute $r = g^k \bmod p \bmod n$,
3. Compute $s = k^{-1}(h + r \cdot x) \bmod n$.

(r, s) is the output signature pair. The verification process uses the public key y to verify if the hash r and s are valid signature pairs for h. Since verification entirely relies on the public key and parameters, its resistance against side channels is not relevant; hence, we omit signature verification throughout this section.

ElGamal. In the ElGamal signature scheme, an alternative to DSA, the first signature pair r is computed similarly, but the second pair is computed as $s = k^{-1}(h - r \cdot x) \bmod (p-1)$.

ECDSA. The Elliptic Curve Digital Signature Algorithm (ECDSA) [191] is an elliptic curve variant of the Digital Signature Algorithm (DSA) [112] in which the prime subgroup in DSA is replaced by a group of points on an elliptic curve over a finite field. The public parameters are an elliptic curve E with scalar multiplication operation $\times$, a point G on the curve, and the integer order n of G over E. The secret key d is a random integer satisfying $1 < d < n-1$, and the public key is $Q = d \times G$. Signature generation for a message hash h is as follows:

1. Choose a random secret k such that $1 < k < n-1$,
2. Compute $(x, y) = k \times G$ and $r = x \bmod n$,
3. Compute $s = k^{-1}(h + r \cdot d) \bmod n$.

(r, s) is the output signature pair.

ECSchnorr. The Schnorr digital signature scheme [293], similar to DSA, can support elliptic curves. Among multiple different standards for Elliptic Curve Schnorr (ECSchnorr), the TPM 2.0 uses the ISO/IEC 14888-3 standard. The key generation for ECSchnorr is similar to ECDSA, but the signing algorithm is slightly different: To sign a message $m \in \{0,1\}^*$,

1. Choose an ephemeral key $k \in \mathbb{Z}_n^*$.
2. Compute the elliptic curve point kG and compute the x coordinate $xR = (kQ)_x$.
3. Compute $r = H(xR \,||\, m) \bmod n$.
4. Compute $s = (k + dr) \bmod n$.

The signature pair is (r, s).

In practice, elliptic curve signature schemes are implemented for a small set of standard curves, which have been vetted for security. The targeted elliptic curves that we will discuss in this chapter are the `p-256` [112] and `bn-256` [27] curves, as supported by TPM 2.0. Applications can use the `bn-256` with ECDSA and ECSchnorr schemes, but it is essential for the elliptic-curve direct anonymous attestation (ECDAA) scheme since ECDAA requires a pairing-friendly curve like `bn-256`.

In DSA, ElGamal, ECDSA, and ECSchnorr, it is critical for k to be uniquely chosen for each signature generation and to remain secret. Exposing one instance of k for a known signature results in a simple key recovery: $d = r^{-1}(s \cdot k - h) \bmod n$. Since k is an ephemeral value, a noisy side-channel attack against k cannot reduce the sampling noise using multiple runs of the attack. However, as discussed in this section, lattice attacks can recover the signing key from partial knowledge of k for many signatures. In Section 6.4.2 and Section 6.4.4, we show that we can recover the entire ephemeral k deterministically in a single trace of the computation of the modular inverse $k^{-1} \bmod n$. Single-trace attacks on signature generation illustrate vulnerabilities even in scenarios where an attacker cannot trigger multiple signature generation operations or only collect a single trace.

6.2.2 Hidden Number Problem and Lattices

Boneh and Venkatesan [45] formulated the hidden number problem (HNP) as the following: Let $\alpha \in \mathbb{Z}_p^*$ be a secret integer. In the hidden number problem, one is given a prime p, several uniformly and independently randomly chosen integers t_i in $\mathbb{Z}_p^*$, and also integers u_i that represent the l most significant bits of $\alpha t_i \bmod p$. The t_i and u_i satisfy the property $|\alpha t_i - u_i| < p/2^l$. Boneh and Venkatesan showed how to recover the secret integer α in polynomial time using lattice-based algorithms with probability greater than $1/2$, if the attacker learns enough samples from the l most significant bits of $\alpha t_i \bmod p$.

Private key recovery using partial information. Key recovery from DSA and ECDSA with partial knowledge of the nonce k can be solved efficiently using lattices [45, 260]. These attacks apply to the case when a few bits are leaked about the nonce for multiple signatures, and the adversary can sample many signatures. Researchers have applied lattice-based algorithms for the HNP to attack the DSA and ECDSA signing algorithms with partially known nonces [151, 259, 260, 285]. As a direct consequence, implementation of these signature algorithms in standard cryptographic libraries is vulnerable when the implementation leaks partial information about the secret nonce through side channels [31, 105, 270, 288]. Garcia et al. [113] demonstrate an attack that recovers the sequence of divisions and subtractions from the binary extended Euclidean algorithm (BEEA) for modular inversion. They observe that this sequence leaks some least significant bits of k and apply a lattice-based key recovery algorithm. Lattice attacks can also solve similar HNP instances to recover private keys for other signature schemes such as EPID in the presence of side-channel vulnerabilities [82]. Ronen et al. [286] connected padding oracle attacks to the HNP. Even subtle implementation flaws that leak the bit length of k are sufficient for multi-trace lattice-based key recovery [53, 82, 248]. In these cases, while the algorithm was implemented with enough care to avoid secret-dependent conditional statements, they leak the bit length by skipping the most significant zero bits of k. In Section 6.2.5, we exploit a countermeasure against this attack to precisely leak the nonce bitlength and recover the secret key using a lattice attack.

There are other variants of the HNP, such as the modular inversion hidden number problem [44] and the extended hidden number problem [147]. We focus on the original HNP, where the attacker learns information about the nonce's most significant bits. A second family of algorithms for solving the HNP is based on Fourier analysis. Bleichenbacher's algorithm [39] was the first to make this connection. Bleichenbacher's Fourier analysis techniques can be augmented with lattice reduction for the first stage of the attack, as shown by De Mulder et al. [83]. Bleichenbacher's original algorithm targets a scenario where each signature only leaks a tiny amount of information. In this scenario, the attacker can query for a vast number of signatures. The De Mulder variant requires fewer signatures, but the above lattice techniques are more efficient in this setting. We use lattice attacks because they are more efficient for the amount of side-channel information we obtain.

Lattice construction. The hidden number problem lattice attacks allow us to recover ECDSA nonces and private keys as long as the nonces are *short*. Since the nonces are uniformly selected from $\mathbb{Z}_n^*$, the k_i will follow an exponentially decreasing distribution of lengths, *i.e.*, half will have a zero in the most significant bit (MSB), a quarter will have the most significant two bits zero, etc. We will refer to this event as two *leading zero bits* or 2 LZBs for short. A randomly selected set of nonces k_i will not be likely to be short, and the lattice attack will not be expected to work. This gap is where side channels prove invaluable to the attacker. Given some side information that reveals the number of MSBs of k_i that are zero, one can filter out the signatures with short nonces, yielding a set of signatures where the k_i are all short [45, 376]. To avoid this vulnerability, constant-time implementations of DSA and ECDSA schemes is crucial.

To mount an attack on ECDSA, we follow the approach of Howgrave-Graham and Smart [151] and Boneh and Venkatesan [45] in reducing ECDSA key recovery to solving the Closest Vector Problem (CVP) in a particular lattice. We can then follow the strategy outlined by Benger et al. [31] and embed this lattice into a slightly larger lattice in which the desired vector will appear as a short vector that can be found using standard lattice basis reduction algorithms like LLL [220] or BKZ [292]. Our first step is to define the target lattice from ECDSA signature samples r_i, s_i and m_i. Consider a set of t signature samples $s_i = k_i^{-1}(H(m_i) + dr_i) \bmod n$; rearranging slightly, these define a set of linear relations

$$k_i - s_i^{-1}r_i d - s_i^{-1}H(m_i) \equiv 0 \bmod n$$

where the nonces k_i and the secret key d are unknowns; we thus have t linear equations in $t+1$ unknowns. Let $A_i = -s_i^{-1}r_i \bmod n$ and $B_i = -s_i^{-1}H(m_i) \bmod n$; we thus rewrite our t relations in the form $k_i + A_i d + B_i = 0 \bmod n$. Let K be an upper bound on the k_i. Now we consider the lattice generated by integer linear combinations of the rows of the following basis matrix

$$M = \begin{bmatrix} n & & & & & \\ & n & & & & \\ & & \ddots & & & \\ & & & n & & \\ A_1 & A_2 & \dots & A_t & K/n & \\ B_1 & B_2 & \dots & B_t & & K \end{bmatrix} \tag{6.2}$$

The first t columns correspond to each of the t relations we have generated, with the modulus n on the diagonal of each of these columns; the weighting factors of K/n and K in the last two columns have been chosen so that the desired short vector containing the secret key will have coefficients all of approximately the same (small) size, and therefore be more likely to be found than an unbalanced vector. In particular, this lattice has been constructed so that the vector $v_k = (k_1, k_2, \dots, k_t, K\alpha/n, K)$ is a relatively short vector in this lattice; by construction, it is d times the second-to-last row vector of the basis, plus the last vector, with the appropriate integer multiple of n subtracted from each column corresponding to the modular reduction in each of the t relations. If this vector v_k can be found, the secret key d can be recovered from the second-to-last coefficient of this vector.

Because this target vector v_k is short, we hope that a lattice reduction algorithm like LLL or BKZ might find it, thus revealing the secret key. The inner workings of these lattice basis reduction algorithms are complex; for our attack, we use them as a black box, and the only fact that is required is that the LLL algorithm is guaranteed in polynomial time to produce a lattice vector of length $|v| \leq 2^{(\dim L - 1)/4} (\det L)^{1/\dim L}$; this is an exponential approximation for the shortest vector in the lattice. In practice on random lattices, the LLL algorithm performs somewhat better. It has been observed to find vectors of length $1.02^{\dim L} (\det L)^{1/\dim L}$ [261]. For the lattices of relatively small dimension we deal with here, the approximation factor does not play a large role in the analysis, but for large dimensional lattices, the BKZ algorithm achieves a better approximation factor at the cost of an increased running time. See Boneh and Venkatesan [45] and Nguyen and Shparlinksi [259, 260] for formal analysis and bounds on the effectiveness of this algorithm.

There are two optimizations of this lattice construction that are useful for a practical attack. The first offers only a minor practical improvement; we can eliminate the variable d by, for example, scaling the first relation by $s_0 r_0^{-1} s_i^{-1} r_i$ and subtracting it from the i^{th} equation to obtain $t-1$ linear relations in t unknowns k_i, $0 \leq i < t$:

$$k_i - s_0 r_0^{-1} s_i^{-1} r_i k_0 - s_i^{-1} H(m_i) + r_0^{-1} s_i^{-1} r_i H(m_i) \equiv 0 \bmod n$$

This optimization has the effect of reducing the lattice dimension by one. Otherwise, the lattice construction is the same, except that we replace the K/n scaling factor in the second-to-last row of the basis matrix with a 1. The second practical optimization is to note that since the k_i are always positive, we can increase the bias by one bit by recentering the nonces around 0. That is, let $k_i' = k_i - K/2$; if $0 \leq k_i \leq K$, we now have $-K/2 \leq k_i' \leq K/2$. This one has the effect of increasing the bias by one bit, which is significant in practice. We give empirical results applying this attack to our scenario in Section 6.2.3.

Modification of the lattice for ECSchnorr. We formulate the problem as in Equation (6.2) by writing

$$A_i = -r_0^{-1} r_i \bmod n \quad \text{and} \quad B_i = s_i^{-1} + s_0 r_0^{-1} r_i \bmod n.$$

At that point, we can apply the lattice-based algorithm precisely as we described earlier.

6.2.3 TPM meet Timing and Lattice Attacks

In Chapter 5, we have established that some TPM implementations leak information about the nonces used for elliptic curve signatures. We show how to use standard lattice techniques to recover the private signing key from this information. By applying lattice attack to the leakage of Intel fTPM, our key recovery succeeds after about 1,300 observations and in less than two minutes. Similarly, we extract the private ECDSA key from a hardware TPM manufactured by STMicroelectronics, which is certified at Common Criteria (CC) EAL 4+, after fewer than 40,000 observations. Our timing attacks have three main phases:

Phase 1: The attacker generates signature pairs and timing information and uses this information to profile a given implementation. Attackers can collect the timing using a remote source, for example, the network round-trip time, or precise local source, as discussed in Section 5.2.1. In this pre-attack profile stage, the attacker knows the secret keys and can use this to recover the nonces. Thus, it has perfect knowledge of the correlation between timing and partial information about the secret nonce k that is leaked through this timing oracle. As explained in Section 5.2.3, in our case, this bias is related to the number of leading zero bits (LZBs) in the nonce, which is revealed by the timing oracle. For the vulnerable TPM implementations in this paper, signing a message with a nonce with more leading zero bits is expected to take less time.

Phase 2: To mount a live attack, the attacker can access a secret-related timing oracle as above and collect a list of signature pairs and timing information from a vulnerable TPM implementation. The attacker uses the signature timing information obtained during the profiling phase to filter out signatures and only keep the signature pairs (r_i, s_i) that have a specific bias in the nonce k_i.

The filtering is performed based on timing thresholds obtained during the profiling stage when secret keys are known, and therefore, nonces can be recovered. This filtering makes it possible for the attacker to find the correlation between the computation time and bias distribution. As long as signatures belonging to different bias classes have non-overlapping parts concerning their generation time, it is possible to define thresholds for filtering signatures with nonces of a specific type of bias.

Phase 3: The attacker applies lattice-based cryptanalysis to recover the private key d from a list of filtered signatures with biased nonces k_i. In the noisier cases, e.g., with timings collected remotely over the network, filtering may not work perfectly, and the lattice attack may fail. In these cases, the attacker can randomly choose subsets of filtered signatures and repeatedly run the lattice attack with the hope of leaving the noisy samples out.

Threat models. We put the components of our attacks together to demonstrate end-to-end key recovery attacks in the TPM threat model. We order the presentation of our attacks from weakest to strongest threat model: **1)** We begin with the most potent adversary, who has system-level privileges with the ability to load Linux kernel modules (LKMs). This adversary uses our analysis tool to collect accurate timing measurements. **2)** We reduce the privileges of the adversary to the user-level scenario in which the execution time of the kernel interface can only be measured from userspace. **3)** We show how key recovery is still possible with an adversary who can simply measure the network round-trip timings to a remote victim.

In all our experiments, we initially programmed the TPM devices with known keys to unblind the nonces and facilitate our analysis. We have also verified the success of attacks on ST TPM and Intel fTPM using unknown keys generated by each device. For this, we used the TPM to generate secret keys that remained unknown to us internally, exported the public key, ran the experiments, and finally verified the recovered private key using the exported public key.

System-Level adversary. In this first attack, we used administrator privileges to collect 40,000 ECDSA signatures and precise timings, as shown in the histogram in Figure 5.11 and filtered the samples to select those with short nonces. We used the execution time to classify these samples into three conjectured nonce length categories based on the observed 4-bit fixed window: those with four, eight, or twelve most significant bits set to zero. We then recovered the nonces, and secret keys using the attacks described in Section 6.2.2, implemented in Sage 8.4 [336] using the BKZ algorithm with block size 30 for lattice basis reduction. We verified the candidate ECDSA private keys using the public key.

Figure 6.14 summarizes the key recovery results for a system-level attacker, using samples obtained via simple thresholding with the filter ranges for 12, 8, and 4 LZBs, as shown in Figure 5.11. For example, to recover samples with 4 LZBs, we filtered signatures that took anywhere from 4.75×10^8 to 4.8×10^8 cycles to generate. For the 4-bit bias, we need 78 signatures to reach a 92% key recovery success probability. For the 8-bit and 12-bit cases, we can reach 100% success rate with only 35 and 23 signatures, respectively. However, we need to collect more signatures in total to generate enough signatures with many LZBs.

For the total number of signature operations, the optimal case turns out to be using nonces with a 4-bit bias. Although we need 78 signatures to attack the 4-bit bias, since each occurs with a probability of $1/16$, it takes only about 1,248 signing operations to have these samples. In our setup on the i7-7700 machine, our collection rate is around 385 signatures/minute. Therefore, we can collect enough samples in under four minutes.

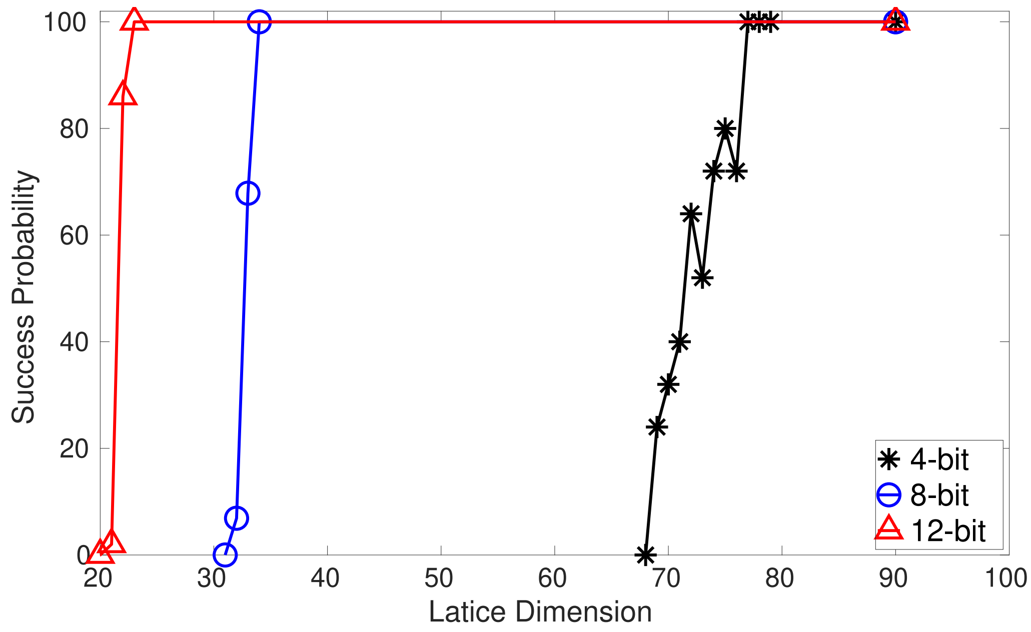


Figure 6.14: **System Adversary:** Key recovery success probabilities plotted by lattice dimension for 4-, 8-, and 12-bit biases for ECDSA (NIST-256p) with administrator privileges.

In the 8-bit case, we need to perform about 8,784 ECDSA signing operations to obtain the 34 appropriate signatures necessary for a successful lattice attack. In total, it takes less than 23 minutes to collect 8,784 signatures. Once the data is collected, key recovery with lattice reduction takes only 2 to 3 seconds for dimension 30, and about a minute for dimension 70. The running time of lattice basis reduction can increase dramatically for larger lattice dimensions, but the lattice reduction step is not the bottleneck for these attack parameters.

Intel fTPM ECSchnorr key recovery: We carried out a similar attack against ECSchnorr by slightly modifying the lattice construction. We were able to recover the key with 40 samples with 8 LZBs. A total of 10,240 signatures were required to perform this attack, which can be collected in about 27 minutes. We also were able to recover the key for the 4-bit case with 65 samples. We obtained these 4-bit samples from 1,040 signing operations that took 1.5 minutes to collect.

STMicroelectronics TPM ECDSA key recovery: We also tested our approach against the dedicated STMicroelectronics TPM chip (ST33TPHF2ESPI) in the system-level adversary threat model. This target is Common Criteria certified at EAL4+ for the

TPM protection profiles and FIPS 140-2 certified at level 2 [321]. It is thus certified to be resistant to physical leakage attacks, including timing attacks [320].

We measured the execution times for ECDSA (NIST-256p) signing computations on a Core i7-8650U machine for 115,000 observations. The machine is equipped with the ST33TPHF2ESPI manufactured by STMicroelectronics. The administrative privileges allowed us to run our custom driver and collect samples with a high resolution.

Following the vulnerability discussion in Section 5.2.3, we began by filtering out any data with execution time below 8×10^8 cycles to eliminate noise. We then sorted the remaining signatures by their execution times. We were able to recover the ECDSA key after generating 40,000 signatures. We recovered the key using the fastest 35 signatures and running a lattice attack assuming a bias of 8 most significant zero bits in the nonces. The required 40,000 samples can be collected in about 80 minutes on this target platform. We are also able to recover the key from 24 samples by assuming 12 LZBs. However, this required generating 219,000 total signatures.

User-level adversary. We now move to a less restrictive model, from a system-level adversary to a user-level adversary where only a user API with user-level privileges is provided to perform the signature operations and measure the execution time. Without the installed kernel measurement tool, we obtain the distribution of signing times shown in Figure 6.15. The noise makes it impossible to distinguish the samples according to the number of leading zero bits in the nonces with high precision. However, we observe that we have a biased Gaussian distribution, and by choosing signatures that have a short execution time, we can still recover the ECDSA key.

We start our analysis by noting that in the system-level adversary setting shown in Figure 5.11, the largest peak is at 4.82×10^8 cycles, while in Figure 6.15 the largest peak is around 4.97×10^8. This behavior is expected since we incur additional latency by measuring the delay from userspace. This noise is independent of the bias, and therefore we set our filtering thresholds by assuming the entire histogram is shifted by moving the profiling measurements to userspace. We collected a total of 219,000 samples. The probability of obtaining a signature sample with 8 LZBs is $1/256$, which means we expect about 855 such signatures among our samples. However, due to the measurement noise, we set a more conservative filtering threshold of 4.76×10^8 cycles and obtained only 53 high-quality signatures. We randomly selected subsets of 30 signatures out of the 53 signatures and ran the lattice reduction 100 times. Experimentally, we observed that it took 34 signatures to recover the key with 100% success rate. Running BKZ with block size 30 for the lattice of this size took 2 to 3 seconds on our experimental machine. After obtaining the key, we recovered the nonces and verified that most of them had the

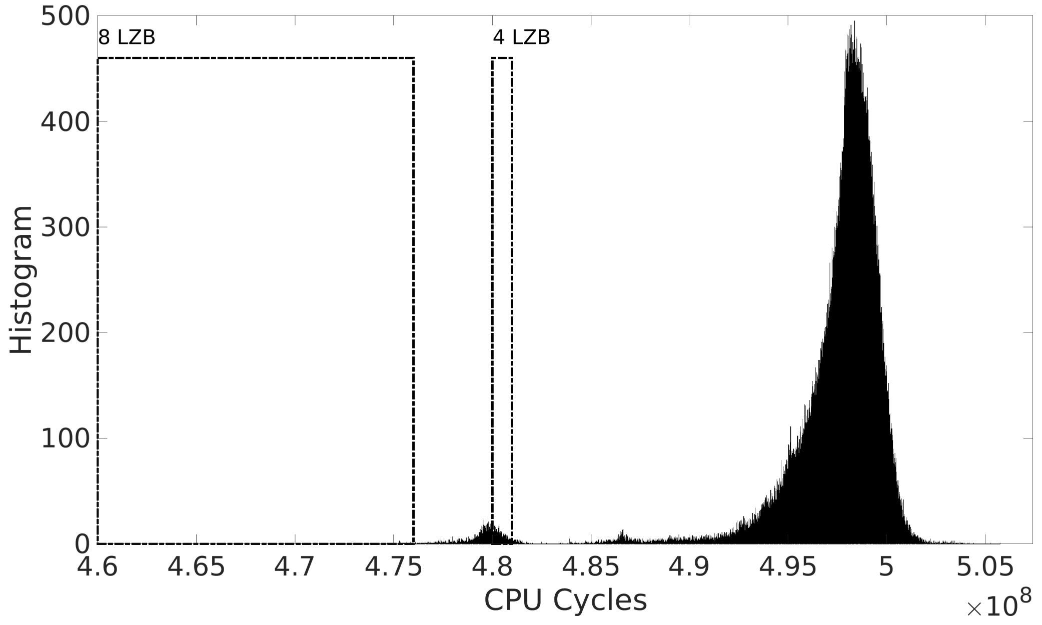


Figure 6.15:]
User Adversary: Histogram of ECDSA (NIST-256p) signature computation times on the Core i7-7700 machine for 40,000 observations. The measurements were collected by a user without administrator privileges.

eight MSBs set to zero[1]. If we had used the entire distribution, we would need about $256 \times 34 = 8,704$ signatures. We use the empirical numbers from our experiments to estimate the likelihood of obtaining such samples in our experimental setup given our choice of thresholds and the noise we experienced; in this case, the probability of obtaining such a sample is 53/855. The estimated total number of signatures required to carry out the attack is 140,413, which takes about 163 minutes to collect. In the 4-bit case, the thresholds we used to filter the samples were between 4.8×10^8 and 4.81×10^8 cycles. With 77 signatures, we recover the key with overwhelming probability. This translates to $77 \times 16 = 1,232$ signatures. But we also need to account for filtering from a narrower range, which results in 1,121 samples out of the $13,687$ expected signatures with 4 LZBs from our total of 219,000 samples. In this case, we estimated that in total, $15,042$ signatures are required for the attack, which takes approximately 18 minutes to collect.

[1]There were few samples with 12 zero MSBs in the analysis

6.2.4 Network Timing Attack on TPM ECDSA

This section further highlights the impact of these vulnerabilities by demonstrating a remote attack against a StrongSwan IPsec VPN that uses a TPM to generate the digital signatures for authentication. We demonstrate a remote attack that breaks the authentication of a VPN server that uses Intel fTPM to store the private certificate key and sign the authentication message. In this attack, the remote client recovers the server's private authentication key by timing only 45,000 authentication handshakes via a network connection.

We demonstrate the viability of **over the network attacks** from clients targeting a server assisted by an on-die TPM. The remote attacks are demonstrated on a simple local area network (LAN) with the attacker and victim workstation connected through a 1 Gbps switch manufactured by Netgear. To this end, we first profile a custom synthesized UDP client/server setup where we can minimize noise. This setup allows the gauging of time measurement for the processes and networking. We later analyze the timing leakage observed by a remote client from a server running StrongSwan VPN software.

Remote UDP attack. We created a server application that uses the Intel fTPM to perform signing operations. The server receives a request for a signature and returns the user's signature over a simple protocol based on UDP. The client (the attacker) sends requests to the server and collects the signatures while timing the request/response round-trip time. Figure 6.16 shows the collected timing information for 40,000 requests. Although there is some noise in the measurement, we can still distinguish signatures generated using short nonces. Figure 6.17 shows our key recovery results.

The experimental results match our expectations outlined earlier since the TPM takes around 200 milliseconds to generate a signature, which is a large enough window to leak timing information over the network. We filtered 8-bit samples by thresholding at 4.93×10^8 cycles and for 4-bit samples at 4.97×10^8 cycles measured on the client. For the case of 4-bit bias, we need 78 signatures above our timing threshold to recover the key, which corresponds to 1,248 signature operations by the server. This can be collected in less than 4 minutes. For the case of 8-bit bias, we recover the key using 47 signatures with high probability, which requires 31 minutes of signing operations. These results demonstrate that remote attacks on fTPM are viable. Next, we explore this direction further by targeting the StrongSwan VPN product.

Remote timing attack against StrongSwan. StrongSwan is an open-source IPsec Virtual Private Network (VPN) implementation supported by modern OSes, including Linux and Microsoft Windows. VPNs can use the IPSec protocol for encryption and

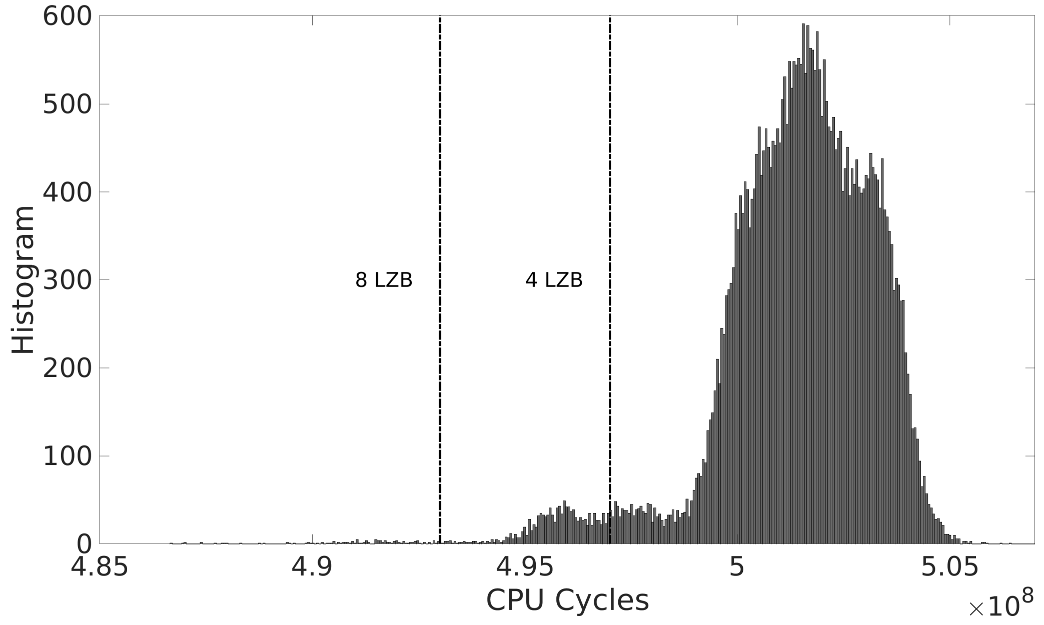


Figure 6.16: Histogram of ECDSA (NIST-256p) signature computation times over the network for 40,000 observations. A server application running on our Core i7-8705G machine is performing signing operations over a simple UDP-based protocol. The client measures the request/response round-trip time to receive a new signature after each request.

authentication. The IPsec key negotiation happens via the IKE protocol, using either pre-shared secrets or digital certificates for authentication. StrongSwan further supports IKEv2 with signature-based authentication using a TPM 2.0 supported device [327]. Here, we attack a StrongSwan VPN Server configured to use the TPM for digital signature authentication by measuring the IKE authentication handshake.

IKEv2 Interleaved Authentication with TPM signatures: We configure our server to use the standard IKEv2 signature authentication with interleaved handshakes where the authentication is performed by an IKE_SA_INIT and an IKE_AUTH exchange between the client and server. Figure 6.18 shows these two handshakes, where the second handshake triggers the TPM device to sign the authentication message. The first exchange of the IKE session, IKE_SA_INIT, negotiates security parameters, sends nonces and performs the Diffie-Hellman Key exchange. After the first exchange, the second exchange, IKE_AUTH, can be encrypted using the shared Diffie-Hellman (DH) key. In the second exchange, the two parties verify each others' identities by signing each others' nonces. We generated a unique ECDSA attestation key (AK) using the Intel fTPM device on the VPN

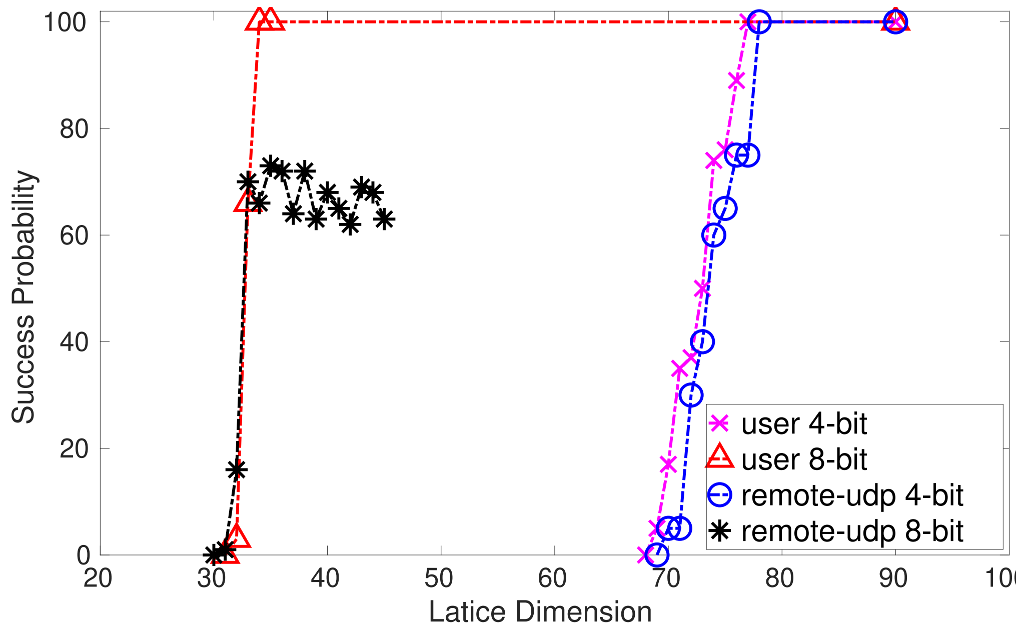


Figure 6.17: **User-Level Adversary and Remote UDP Attack:** Key recovery success probabilities by lattice dimension for 4-bit and 8-bit cases for ECDSA (NIST-256p) with timings collected from the userspace in one scneario, and over the network from a remote client in another scenario.

server. The TPM device only exposes the public portion of the AK. Then we generated a self-signed attestation identity key (AIK) certificate and stored the ECDSA AIK certificate in the non-volatile memory of the TPM device. During the second exchange, the server asks the TPM device holding the private AK to sign the client's nonce and return the signature to the client. When the client receives the signature, she can verify that her nonce is signed with the legitimate server's AK corresponding to the AIK certificate. However, a malicious remote client or a local user who can exploit the timing behavior to recover the private AK can forge valid signatures and act as a legitimate VPN server.

StrongSwan VPN key recovery: As a malicious client, we perform the following steps to collect timing measurement and recover the secret AK:

1. The malicious client performs the first handshake with the server to exchange security parameters, nonces and completes a Diffie-Hellman exchange.

2. The malicious client starts a timer and initiates the second handshake. After the server signs the client's nonce and other security parameters using the TPM device, the malicious client will receive the signature and measure the total handshake time. We discovered the TPM signature timing vulnerability might delay this exchange

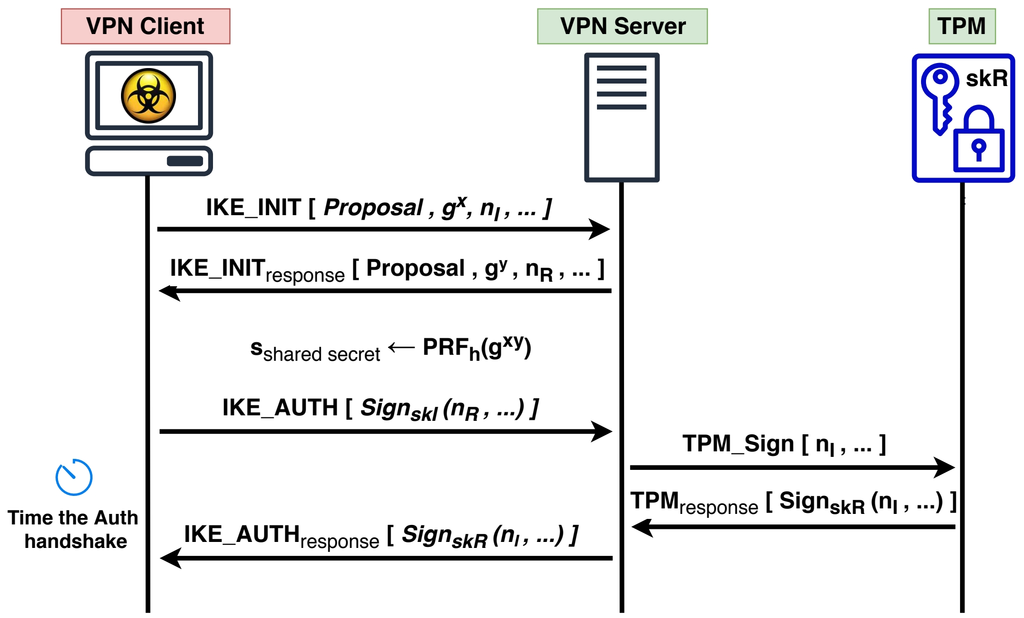

VPN Client
VPN Server
TPM
skR
IKE_INIT [Proposal , g^x, n_I , ...]
IKE_INIT$_{response}$ [Proposal , g^y , n_R , ...]
$s_{shared\ secret} \leftarrow PRF_h(g^{xy})$
IKE_AUTH [$Sign_{skI}$ (n_R , ...)]
TPM_Sign [n_I , ...]
TPM$_{response}$ [$Sign_{skR}$ (n_I , ...)]
IKE_AUTH$_{response}$ [$Sign_{skR}$ (n_I, ...)]
Time the Auth handshake

Figure 6.18: Steps of IKE_SA_INIT and IKE_AUTH exchange between the client and server running StrongSwan VPN.

based on the nonce used in the signature generation, leaving an observable effect on network packet timings.

3. The malicious client stores the network timing and the received signature pairs and discards the session by sending an `IKE_INFORMATION` packet to the server. It repeats this process, starting from the first step to collect enough time measurements and signatures.

To determine if there is any exploitable leakage observed over the network, we collected both remote timings on the client and local timings on the server running a StrongSwan VPN software on our Core i7-8705G machine, where an Intel fTPM computes ECDSA signatures. The histograms for 40,000 timing measurements observed both locally and on the server are shown in Figure 5.11 and Figure 6.19.

The identifiable separate peaks corresponding to 4-bit and 8-bit leakage in Figure 5.11 are no longer observable with measurements collected over the noisy network in Figure 6.19. Still, the relative location of the peaks in the local timings histogram can be used as a template to design filters to be applied on the remote timings. For this, we need to account for the change in clock frequencies. As a simple heuristic, we scale the filter ranges in Figure 5.11 by the ratio of the time when the gigantic peaks are observed, *i.e.*, $3.41/4.82$. We also adjusted the filters to account for the additional delay due to remote measurements. Finally, we reduced the widths to cover the left half of the distributions

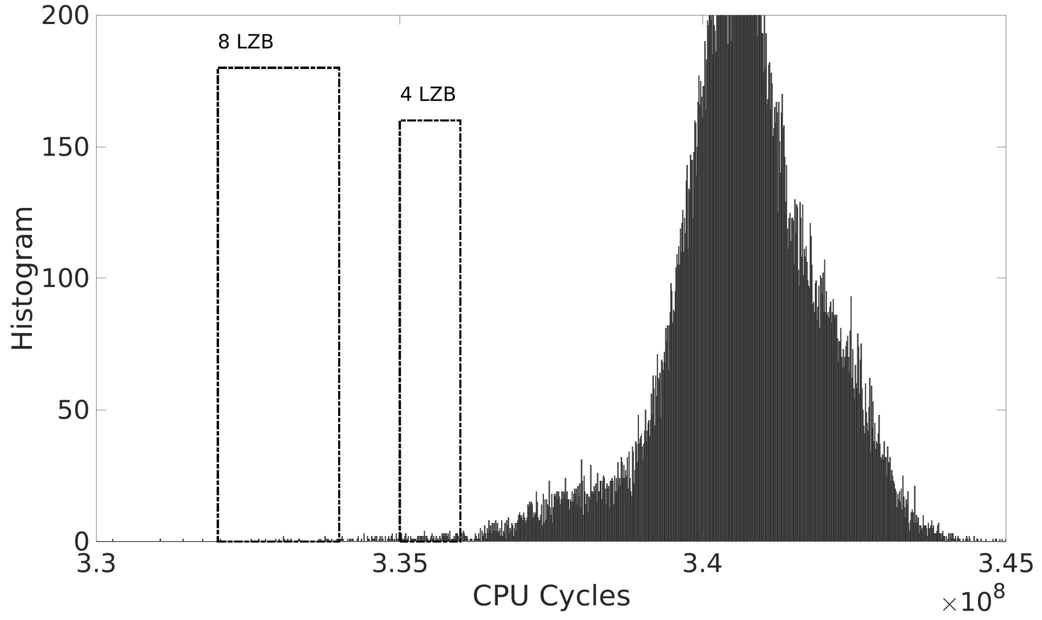


Figure 6.19: Histogram of ECDSA (NIST-256p Curve) signing computation times over the network for 40K observations. The server is running StrongSwan VPN software equipped with Intel fTPM. The client application measures the request/response round-trip time.

since they yield cleaner samples. For 8-bit samples, we filter between 3.32×10^8 and 3.34×10^8, and for 4-bit 3.35×10^8 and 3.36×10^8, obtaining 153 8-bit and 222 4-bit samples. We then applied the lattice attacks from Section 6.2.2 to these samples using our Sage implementation and BKZ-2.0 reduction with block size 30 over many iterations. Figure 6.20 shows the results. We recover the key with high probability for both the 4-bit and 8-bit cases after dimensions 34 and 80, respectively. In the 4-bit case, we used 222 out of the expected $1/16 \times 198K = 12,375$ 4-bit samples. To end up with 80 4-bit samples, we would need to sample $80 \times 16 = 1,280$ times. However, since we are filtering for high-quality samples within the nonces with 4-bit bias with probability $222/12,375$, we need also to take that into account. This means we need about $1,280 \times 12,375/222 = 71,351$ signatures. The 8-bit case used 153 out of the 774 expected 8-bit samples. This result means we need about $34 \times 256 = 8,704$ samples. Accounting for filtering with probability $153/774$, we need about $8,704 \times 774/153 = 44,032$ signatures. In this case, targeting the nonces with 8-bit bias turns out to be more efficient. The noise introduced by measuring remotely on the client side has rendered 4-bit samples harder to distinguish, and therefore these require more aggressive filtering. We can collect about 139 signatures per minute from StrongSwan. This analysis means we can collect enough samples in about 5 hours 16 minutes. Compared to the local case, the increased attack time is due to the delay

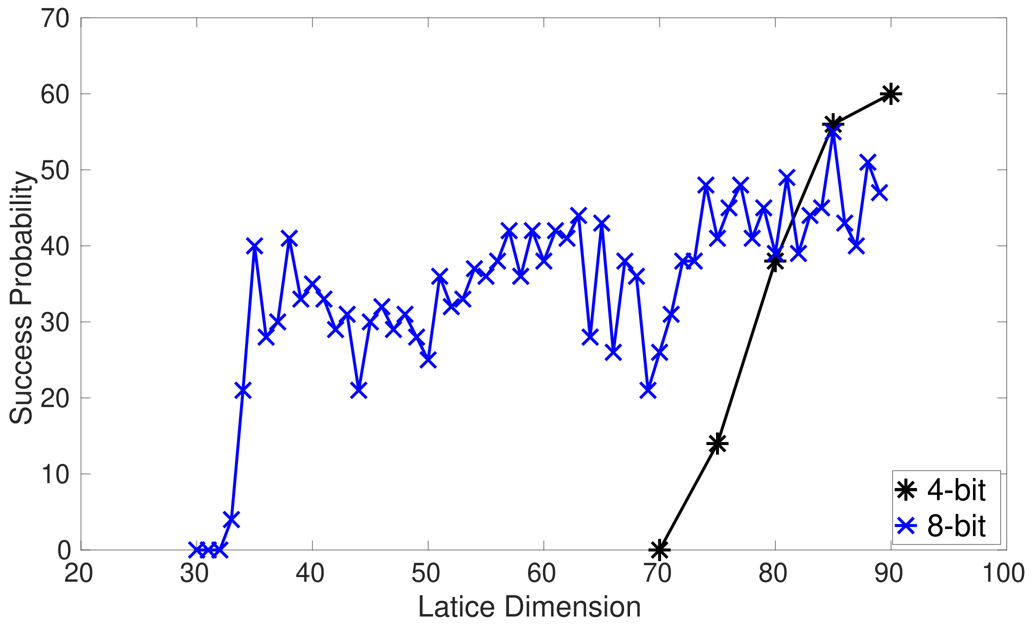


Figure 6.20: **Remote StrongSwan Attack:** Key recovery success probabilities by lattice dimension for the 4-bit and 8-bit cases for ECDSA (NIST-256p) with samples collected on the client.

incurred during the handshakes: it takes approximately $855/139 = 6.15$ times longer to collect each signature.

In our attack, we queried the VPN server directly to collect the signatures and timings. This attack can also be performed by an active man-in-the-middle (MiTM) adversary who hijacks a DH key exchange. However, there is no additional benefit to be gained over the malicious client since the attacker is active in both scenarios. A passive attack would not be possible since the signatures are encrypted with the shared secret between the client and the server. Another critical factor that affects the viability of the attack is networking noise. Depending on the type and traffic of the network, e.g., networks with high bandwidth, local organizational networks, and local private networks on the cloud, the attack's success rate will vary. Typically in cloud environments, network connections between cloud nodes tend to have higher bandwidth and more stable connections and will have less timing noise.

6.2.5 Breaking ECDSA Timing Protection in SGX

WolfSSL uses the subroutine `wc_ecc_mulmod_ex` (Listing 7) to compute the scalar multiplication $k \times G$ while generating the signature. This subroutine has built-in mitigations against side-channel attacks and implements an `always-add-and-double` algorithm by

Threat Model	TPM	Scheme	#Sign.	Time
Local System	ST TPM	ECDSA	39,980	80 mins
Local System	fTPM	ECDSA	1,248	4 mins
Local System	fTPM	ECSchnorr	1,040	3 mins
Local User	fTPM	ECDSA	15,042	18 mins
Remote SSwan	fTPM	ECDSA	44,032	$\sim$5 hrs

Table 6.2: Summary of our key recovery results.

arithmetizing the conditional check for the add. As a result the scalar operations add at Line 15/18 and double at Line 16/19 will both be executed for all scalar bits. This prevents an adversary from learning the nonce k bit by bit. The second countermeasure that is implemented in this implementation aims to protect against attacks exploiting the bitlength of the nonce [53, 248]. This countermeasure executes a sequence of dummy operations for each leading zero bit. While these dummy operations mitigate side channels like data cache attacks, page-level attacks, and timing attacks, we can use CopyCat to distinguish the branch outcome at Line 13 and leak the bit length of nonce k. In this section, we show that when an implementation leaks partial information [113], CopyCat maximizes the capability of the attacker as if the leakage in the implementation were introduced as a constant bias [259].

Recovering dummy operations. We analyze `wc_ecc_mulmod_ex` using CopyCat. In this analysis, we count the number of instructions executed between consecutive accesses to the page that holds the `ecc_projective_dbl_point` subroutine. The trace shows that for one transition of basic blocks, we can observe 49 steps when the function is processing the dummy operations. As soon as the subroutine switches to the real operations, this step count will change to 46. As a result, we can use this information to determine the number of dummy executions of the `always-add-and-double` sequence from a set of traces. Since we only need to observe the first few bits to recover the nonce's length, we shortened our trace collection to observe only the first 7 bits.

Lattice attack using the nonce bit length. We generated many signature traces, recovered the nonce lengths, and filtered for signatures with short nonces [45]. We followed the approach of Howgrave-Graham and Smart [151] and Benger et al. [31] to formulate the key recovery as a lattice problem. The detailed construction of this lattice is given in Section 6.2.2. The desired short vector of this lattice can be found using

LZBs	Dim	L-Time	Signatures	IRQs	T-Time
4	75	30 sec	1,200	3.9M	13.3 sec
5	58	5 sec	1,856	6.0M	20.4 sec
6	46	3 sec	2,944	9.6M	33.7 sec
7	42	2 sec	5,376	17.5M	1 min

Table 6.3: Minimum number of signature samples for each bias class to reach 100% recovery success for the lattice-based key recovery on `wc_ecc_mulmod_ex` of ECDSA, with lattice reduction time L-Time and trace collection time T-Time.

standard lattice basis reduction algorithms like LLL [220] or BKZ [292], which leads to full recovery of the private key.

Evaluation. We executed this attack for 10,000 signing operations. Our attack recovered the number of leading zero bits with 100% accuracy. On average, each attack issues 3244 IRQs to count 2542 steps of the scalar multiplication operation. Section 6.2.5 shows the results for key recovery using various nonce bit lengths. Since the nonce length is recovered without noise, the lattice attack is quite efficient.

6.3 Template MDS Attack on Constant-time RSA

In this section, we demonstrate the attack potential of Medusa by extracting an RSA key from OpenSSL. We show that Medusa can leak various parts of an RSA key during the `base64` decoding stage. By leaking various smaller chunks from an RSA-1024 private key, we can apply a lattice-based cryptoanalysis technique to recover the entire key within 20 minutes. Then we build leakage templates and recover full RSA keys by employing lattice-based cryptanalysis techniques.

6.3.1 RSA Cryptosystem

RSA keys [284] are generated as follows:

1. Choose large prime numbers p and q, compute $N = pq$,
2. Compute the least common multiple $\lambda(N) = \mathrm{lcm}(p\text{-}1, q\text{-}1)$,
3. Choose e such that $1 < e < \lambda(N)$ and $gcd(e, \lambda(N)) = 1$,
4. Compute $d = e^{-1} \bmod \lambda(N)$.

(N, e) are public and $(p, q, \lambda(N), d)$ are private. RSA implementations commonly use the Chinese remainder theorem (CRT) to reduce computation time, and generate additional private values $d_P = d \bmod (p-1)$, $d_Q = d \bmod (q-1)$, and $q_{inv} = q^{-1} \bmod p$. A signature is the value $s = h^d \bmod N$ where h is a hashed and padded message. Signature verification checks if $h \equiv s^e \bmod N$.

To prevent side-channel attacks on signature generation, most implementations blind the input h with a random r before computing the modular exponentiation: $s_b = (hr^e)^d \bmod N = h^d r \bmod N$. Later, the unblinded signature can be computed as $s = s_b r^{-1} \bmod N$. As a result, attacks on RSA key generation have gained recent attention [14, 17]. However, since the private key parameters are only computed once, an attack against RSA key generation must only require a single trace.

6.3.2 Sampling Partial RSA Secrets from OpenSSL

In Section 3.3.3, we analyzed the occurrence of `rep mov` in popular cryptographic libraries. For the attack, we focus on OpenSSL 1.1.1c, as it is both widely used and it deploys countermeasures against traditional side-channel attacks, making it a robust target. Note that while we did not analyze other cryptographic libraries further, we expect that they are vulnerable to the same or a similar attack as well. As the victim, we use a simple artificial application that leverages OpenSSL to load an RSA key from a file and signs some data using this key. In our attacker model, we can start the application arbitrarily often, but we do not control any inputs to the application. Note that this scenario, *i.e.*, triggering the victim application, is in line with previous research [61, 300, 356, 384].

Every time the application is started, it has to load the RSA private key from the key file. The key file is in the PEM format, a `base64` encoded representation of the key parameters. Hence, to load the actual key into memory, OpenSSL first has to decode the key file using its internal `base64` decoder. When compiling the library to optimize it for size, the `base64` decoder uses `rep mov` for loading the `base64`-encoded key from the key file. We attack exactly this `rep mov` sequence using MEDUSA to leak the RSA parameters, which are required to derive the private key.

Running Medusa. OpenSSL keys in PEM format include both the default prime and exponents of the RSA alongside the precomputed parameters for the Chinese Remainder Theorem (CRT) computation. This includes modulus N, public exponent e, private exponent d, prime numbers p and q, $d \bmod (p-1)$, $d \bmod (q-1)$ and the coefficient $q^{-1} \bmod p$. The size of the copy operation during the execution of the `rep mov` instruction depends on the key size. For example, for a 1024-bit RSA key, there are $5*64+2*128 = 576$ bytes of key material to be copied. As the key material also includes several bytes for the

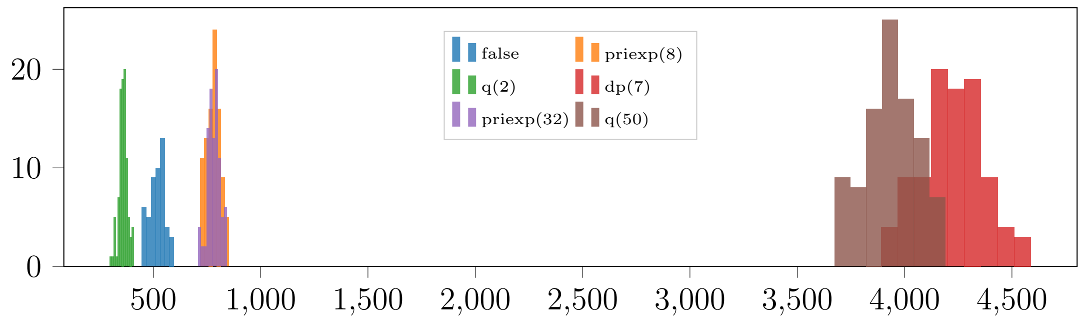


Figure 6.21: Histogram and score of most likely 6-byte leakages through AVX256-P3 with 10K observations collected in 100 runs (labeled by starting bytes). Six byte block leakages at `q(2)` (q starting at byte 2), `priexp(8)` and `priexp(32)` (RSA exponent d starting at bytes and 8 and 32) and `dp(7)` (leak from $d_p = d \bmod p - 1$ starting at byte 7) can be easily identified based on the observation frequencies.

ASN.1 PEM metadata, the total amount of copied raw data is approximately 600 bytes. Since the data is `base64` encoded, which always encodes three raw bytes as 4 bytes, the actual amount of copied information is approximately 800 bytes. Hence, depending on the copy operation's size and the attack employed, different parts of this key may be leaked more often (cf. Figure 3.3).

We create a template attack based on the frequency of different parts of the entire RSA secrets' leakage to tackle this limitation. In this attack, we use Variant 2 of MEDUSA to leak the data with the unaligned store forwarding, revealing the common data bus's entire content. We also use the domino technique [300] combined with the frequency of each observed value to build a frequency template of recovered key parts. As discussed in Section 3.3.1, the probability of leaking specific data depends on the offset of the leaked data transmitted over the common data bus. Hence, depending on which part of the data we want to leak, we have to repeat MEDUSA between 10 000 and 20 000 times per key byte. In total, we run this experiment 100 times. Based on the frequency of an observed 8-byte block of `base64`-encoded data, we can create a template that tells us which parts of the key material are leaking more often. Note that each 8-bytes block of `base64` encoded key material holds 6-bytes of raw key parts. Figure 6.21 and Figure 6.22 show the frequency of each section leaked through different part of an AVX-256 register. Note that in the top histogram, we see consistent, substantial leakage of 6-byte blocks in `priexp` (the RSA key d), starting at byte locations $14, 38, 86$, and 110 as well as substantial leakage in q starting at locations $8, 32, 56$. The dark grey pieces show the information recovered that does not belong to the RSA parameters. Note that the histograms only show the most dominant leakages to prevent crowding in the presentation.

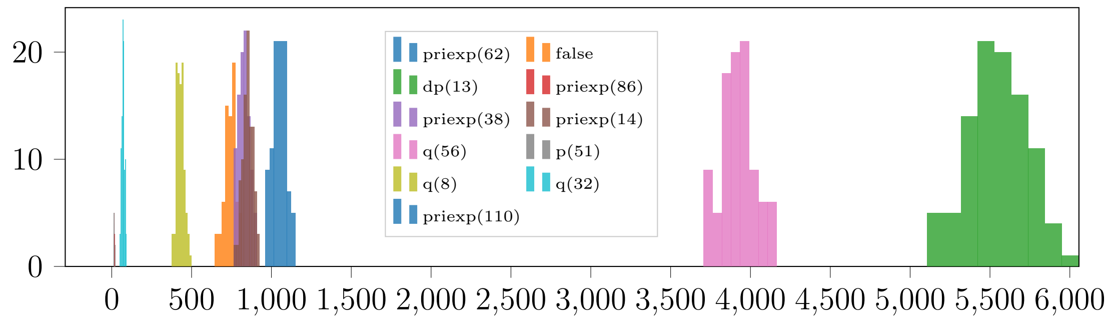


Figure 6.22: Histogram and score of most likely 6-byte leakages through AVX256-P4 (similar experiment as Figure 6.21). Block leakages at `q(8)`, `q(32)`,`q(56)` (q starting at bytes 8, 32, 56), `priexp(14)`, `priexp(39)` and `priexp(86)`, `dp(13)` (leak from d_p starting at byte 13), `p(51)` (p starting at byte 51) can be identified based on the block frequencies.

6.3.3 Recovering full RSA keys using Lattice Attacks

These leakages give us only **partial** information on the RSA secrets p, q, d (`privexp` in the OpenSSL implementation), and $d \bmod (p-1)$, $d \bmod (q-1)$ and the coefficient $q^{-1} \bmod p$ are far from yielding the full secrets. However, we have seen significant progress in recovering keys from RSA instantiated with partially exposed messages or decryption keys. Coppersmith introduced a technique for finding small roots of polynomial equations by reducing the problem of finding roots of a polynomial $f(x)$ over $\mathbb{Z}_p$ [76]. Cryptanalysis can benefit from this technique to recover RSA factors if the least or most-significant half of the bits of p or q are known. Boneh, Durfee, and Frankel proposed a technique to recover the RSA secret and moduli p and q if a quarter of the least or most significant bits of d are known when e is small enough to reach via exhaustive testing [43]. Later Boneh and Durfee [42] presented a technique that recovers RSA factors with $d < N^{0.292}$ without any conditions on e. For an overview, see May [234], and the more recent Takayasu and Kunihiro et al. [329]. Here we focus on two attacks which fit our leakage profile:

Coppersmith. We use the Coppersmith attack to recover the RSA factor q. We combine partial leakages of q at bytes 8, 56 (from P4), and 2, 50 (from P3) and 0, 61, 12, 44 (from P2) to obtain a leakage in q: 18-bytes LSB (bytes 0-17) and 20-bytes MSB (bytes 44-63). This combined leakage gives us information about more than a quarter (38/128 bytes) of N for the 1024-bit RSA. Coppersmith's attack is slightly adjusted to handle the LSB/MSB split into the leaked data. We apply Coppersmith's lattice attack to recover small solutions to

$$f(x) = x + (q_{MSB}2^{44\times 8} + q_{LSB})(1/2^{18\times 8} \bmod N) \ .$$

We used `SageMath v8.4` with `NTL` for LLL to implement the attack which takes a few second to successfully recover a root x_0 and the RSA factor: $q = q_{MSB}2^{44\times 8} + x_0 2^{18\times 8} + q_{LSB}$. We attached *scores* by counting how many times the partial leakages could be stitched together into an 8-byte block over 20 000 samples. The scores, as shown in Section 6.3.3 serve as a template which we use to classify observations before trial by Coppersmith. To obtain the statistics for the templates, we need 20 000 observations. With more spurious blocks (selected as to have a score within ±20 % of the target block), we need to try more combinations. On average, we need 58 000 trials.

	ymmX-P2				ymmX-P3		ymmX-P4	
Block `q(i)`	44	12	61	0	2	50	8	56
Avg. Score	82	288	304	355	377	4157	401	3651
# Spurious	5	18	16	14	0	1	0	0

Table 6.4: Leakage scores used for the template Coppersmith attack (q prime).

Boneh, Durfee, Frankel (BDF). The BDF attack [43] recovers RSA factors given the LSB quarter of the secret exponent d bits when e is small enough to be exhaustively tested. The attack iterates the following steps for each $k \in [1, e]$ until a solution is found:

1. Form a polynomial equation:

   $$f(x) = kx^2 + (ed_0 - k(N+1) - 1)x - kN = 0 \pmod{2^{n/4}} \ .$$

   Here $n = \log_2(N)$ and $d_0 = d \pmod{2^{n/4}}$.
2. Find solutions to $f(x)$. Due to the special structure of the modulus, the equation is efficiently solved to recover at most 2^{t+1} solutions, where t is the largest power of 2 that divides k. For correctly chosen k the solution of $f(x)$ yields p (or q) modulo $2^{n/4}$.
3. Check each recovered solution by taking it as the (candidate) LSB of p or q and running Coppersmith to see if we obtain the RSA factors.

The algorithm runtime is $\mathcal{O}(e \log(e))$ Coppersmith iterations.

A small but effective optimization. Our target $e = 2^{16} + 1$ is exhaustible. However, we can do much better since we have some LSB bytes of p and q. We can use these bytes to check the recovered candidate LSBs of p or q and take a shortcut omitting costly Step 3 if there is no match. With a few bytes of leakage, we can reduce the complexity from $\mathcal{O}(e\log(e))$ to only $\mathcal{O}(\log(e))$ Coppersmith evaluations. For the 1024-bit case, we exploit the leakage observed on d (`priexp`) with 6-byte leakages starting at bytes: $2, 8, 14, 16, 26$, which gives us 27 LSB of the required 32 bytes of d. We are missing 5 bytes, which is now exhaustible. The attack requires about 180 trials to cope with the spurious blocks.

	ymmX-P2			ymmX-P3	ymmX-P4
Block $d(i)$	2	16	26	8	14
Avg. Score	116	104	138	739	724
# Spurious	9	8	0	1	0

Table 6.5: Leakage scores used for the template Coppersmith attack (d private key).

Scaling the attack to 2048-bit RSA. The 1024-bit RSA attack described above recovered the secret key using a simple univariate formulation via Coppersmith's technique since a quarter of contagious private key bits were available. The 2048-bit is more challenging since we cannot obtain 64 contagious bytes of q, p, or d through the leakage channel. On the bright side, we have significantly more leakage from the higher blocks of d and non-contiguous blocks of p and q. The main idea is to form *multivariate* expressions of the form $f_i(x, y)$ using the known parts of d, p, and q where x and y represent the unknown parts of p and q. Then we apply lattice reduction to reduce the size of the coefficients. A *resultant* computation applied on the reduced multivariate polynomials yields a univariate polynomial, whose solution will produce the unknown parts of p or q. The success probability for the attack depends on the amount of leakage and the precise lattice formulation. While plausible, this approach is beyond the scope of this paper. For further information on multivariate analysis see [40, 102, 117].

6.4 Single-trace Attacks on Public Key Schemes

Public-key algorithms that execute variable operations for each bit of a secret input, like the square-and-multiply algorithm for modular exponentiation and scalar multiplication based on Montgomery ladders, are susceptible to side-channel leakage. Earlier attacks exploit such algorithms [383, 384, 390] where the victim is triggered many times to compensate for potential sampling noise. These attacks generally conclude with the recovery of most of the secret bits. Nowadays, most implementations have adopted constant-time algorithms like fixed-window scalar multiplication to mitigate such attacks [282]. To demonstrate the strength of CopyCat, we develop single-trace attacks that allow efficient cryptographic key recovery from multiple widely-used cryptographic libraries.

Previous single-trace attacks on RSA. Recent work has demonstrated a single-trace side-channel attack against RSA key generation that leaks the sequence of divisions and subtractions from the BEEA during the coprimality test $\gcd(e, p-1)$ [16, 365] or secret key generation $d = e^{-1} \bmod \lambda(N)$ [58]. These attacks recover the secrets $(p-1)$ or $\operatorname{lcm}(p-1, q-1)$ from this sequence when e is small enough to be brute forced, which is typically the case in practice[2]. The proposed mitigation is to increase the size of the input e by masking it with a random variable that may be hard coded [58]. In Section 6.4.3, we use CopyCat to recover all the branches from BEEA, not just the sequence of divisions and subtractions. We propose a novel algorithm that uses this information to recover the private factors p and q from $e^{-1} \bmod \lambda(N)$. Our attack works even for large e, thwarting the above mitigations.

Furthermore, our algorithm can recover the key from a modular inversion algorithm with multiple unknowns. We demonstrate a novel end-to-end single-trace attack on the CRT computation $q^{-1} \bmod p$. In a concurrent and independent work, Aldaya et al. [14] outline a different key recovery algorithm for $q^{-1} \bmod p$, which is not always successful. Our single-trace attacks on RSA in Section 6.4.3 use a branch-and-prune algorithm inspired by Heninger and Shacham [145]. Bernstein et al. applied a variant of branch-and-prune algorithm to recover RSA keys from a sliding-window modular exponentiation implementation [33]. Similarly, Yarom et al. demonstrated an attack with intra-cache line granularity on a fixed-window implementation of modular exponentiation that recovers a fraction of the bits [385]. In Section 6.4.4, we generalize our attack to implementations of BEEA used in other popular cryptographic libraries. We demonstrate attacks against $gcd(p-1, q-1)$ in OpenSSL X.931 RSA and $q^{-1} \bmod p$ and $e^{-1} \bmod \lambda(N)$ in WolfSSL and Libgcrypt.

Contribution. We extend the cryptanalysis of the binary Euclidean algorithm used for modular inversion in most common libraries we examined. We propose novel algorithms for efficiently recovering cryptographic keys from a single control flow trace for DSA and ECDSA digital signature generation and RSA key generation. The libraries we examined implemented numerous mitigations against side-channel attacks, including always-add-and-double for elliptic curve scalar multiplication and RSA exponent masking. Still, these protections were insufficient to protect against CopyCat. We conclude that new classes

[2] e is commonly chosen as $2^{16} + 1 = 65537$.

of defenses will be necessary to protect against this type of high-granularity, deterministic, and noise-free attack.

Consequently, CopyCat enables a class of side-channel cryptanalysis targeting secure enclaves that have previously not been sufficiently understood. In an extensive study of cryptographic implementations within widely-used libraries, we discover that most of them have exploitable flaws one way or another when the leakage model matches the capability of CopyCat. On top of this, we propose new single-trace attacks that apply to multiple signature schemes [112] and attacks on RSA key generation based on the branch-and-prune algorithm [145]. We demonstrate how combining these techniques with the leakage obtained by CopyCat allows efficient single-trace key recovery in multiple real-world implementations. Our practical attacks on Intel SGX is deterministically reproducible without additional assumptions or sampling noise. We hope that our findings will help developers and the community to improve protection against this class of vulnerabilities.

6.4.1 Unleashing CopyCat on WolfSSL

WolfSSL is a prominent, FIPS-certified solution officially supporting Intel SGX [374]. In a case study on the WolfSSL cryptographic library, we show that CopyCat enables attacks that were not previously possible without a deterministic and fine-grained leakage model. we outline our controlled-channel attack using CopyCat to precisely recover the full execution trace of WolfSSL's implementation of the binary extended Euclidean algorithm (BEEA) used for modular inversion of cryptographic secrets in DSA, ECDSA, and RSA. Precise recovery of the full execution flow of BEEA enables new single-trace algorithmic attacks on both DSA signing and RSA key generation, as demonstrated in Sections 6.4.2 and 6.4.3, respectively. Finally, we apply CopyCat to bypass incomplete side-channel mitigations and recover deterministic partial information on ECDSA signatures, which allows for efficient key recovery via lattices.

Experimental setup. Our experimental setup includes a desktop Intel Core i7-7700 CPU that supports Intel SGX equipped with the latest microcode (0xca) running Ubuntu 16.04 with kernel 4.14.0-72-generic. We use the SGX-Step [351] framework v1.4.0 to implement our attacks on the latest stable WolfSSL version 4.2.0. WolfSSL officially supports compilation for Intel SGX enclaves. We implemented our key recovery attacks in SageMath version 8.8.

CopyCat analysis of BEEA. Computing the modular inverse or greatest common divisor (GCD) using the binary extended Euclidean algorithm (BEEA) has previously exposed cryptographic implementations to side-channel attacks [7, 113, 365]. The BEEA, as shown in Algorithm 3, is not constant time and can leak various bits of its input. However, previous attacks are limited to recovering only partial and noisy information about the secret input. This limitation stems from low spatial resolution and the presence of noise. For instance, a cache- or page-level attacker who can distinguish which arithmetic subroutines the algorithm has invoked cannot determine the outcome of the comparison at line 13; both directions of the branch generate the same sequence of memory access patterns. Additionally, the arithmetic functions may fit within the same page and become indistinguishable for a page-level adversary. Alternatively, a cache attacker may track these branches' outcomes within the same page by monitoring the corresponding instruction cache lines for the BEEA subroutine. However, a compact implementation of this algorithm can fit multiple branches within the same cache line. While some microarchitectural attacks on the instruction stream may leak some of these low-level branch outcomes, they are all prone to various amounts of noise [15, 104, 153, 218, 352].

WolfSSL supports subroutines `fp_invmod_slow` and `fp_invmod`[3] as two different BEEA implementations. The former is a straightforward implementation, and the latter is a compact implementation that only supports odd moduli. We analyze both implementations and show how to use CopyCat to recover these implementations' runtime control flow deterministically and without noise.

Binary layout. The subroutines `fp_iseven` and `fp_isodd` are simply inlined within the same page as their caller `fp_invmod_slow` after the compilation. However, the arithmetic functions A=`fp_add`, C=`fp_cmp`, D=`fp_div_2`, and S=`fp_sub` are external calls and reside in a new page. Analyzing these arithmetic functions (A, C, D, S), including their internal subroutines, shows that they span $2,895$ bytes. Hence, it is reasonable to assume that they can fit into a single 4 KiB page, thus preventing a page-level attacker from distinguishing them at runtime altogether. Besides, even assuming they do not align within the same page, reconstructing the exact execution flow is impossible. For example, the transition from S to D can result from multiple different code paths. The instructions for `fp_invmod_slow` can fit into fewer than six cache lines with multiple basic blocks[4] overlapping within the same line.

WolfSSL also supports a modified version of BEEA, `fp_invmod` specialized to the case of odd modulus, which is used for RSA $q^{-1} \bmod p$ (§6.4.3) and DSA $k^{-1} \bmod n$ (§6.4.2). The control flow and overall layout for `fp_invmod` are similar to the above implementation, but it is more compact, as it does not include some of the arithmetic

[3] `fp_invmod_slow` and `fp_invmod` can be found at line 885 and 1015 of `https://github.com/wolfSSL/wolfssl/blob/48c4b2fedc/wolfcrypt/src/tfm.c`, respectively.

[4] A basic block is a code sequence that has no branches in and out.

statements. `fp_invmod` can fit into fewer than four cache lines with multiple overlapping basic blocks.

Recovering BEEA control-Flow transfers. We analyzed the runtime control flow of `fp_invmod_slow` by matching its disassembly with the execution trace we recovered from running CopyCat. Figure 6.23 shows the control flow transfers at page-level granularity for the page corresponding to `fp_invmod_slow` and the page corresponding to arithmetic functions (Circles). Additionally, each arrow's weight shows what number of instructions `fp_invmod_slow` execute before accessing the page corresponding to arithmetic functions. The division loop for u_i (*u-loop*) and v_i (*v-loop*) have a similar control flow. In addition, the two blocks of substitutions after the comparison of $u > v$ have similar control flow for both the left *S1* and right *S2* direction. Only specific transitions are viable from these blocks to division loops during the computation of the modular inverse. For example, *S2* always goes to *v-loop* and *S1* always goes to *u-loop*. Since these instruction counts are distinguishable for transitions related to conditional statements, we can use a trace consisting of a vector of these weights in the graph to infer the conditional statement's outcome.

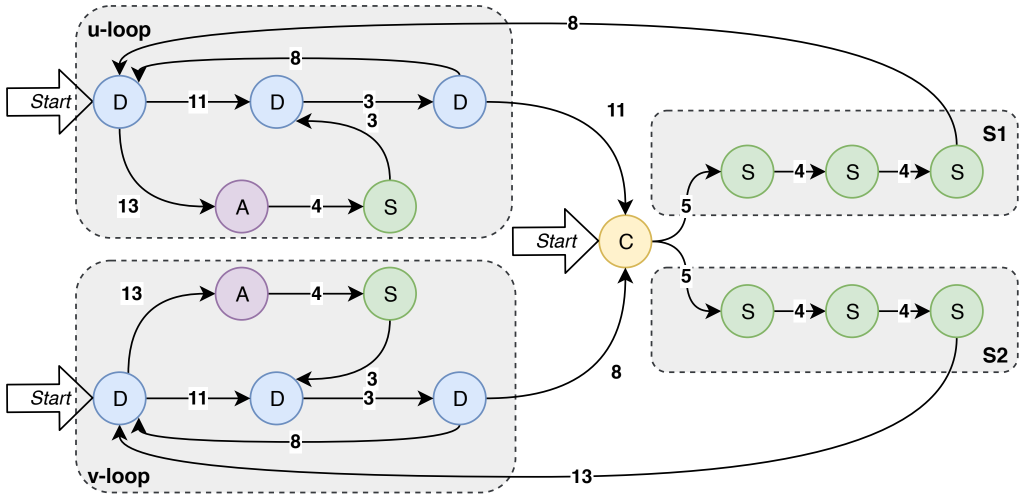


Figure 6.23: Control flow of the BEEA as implemented by `fp_invmod_slow`. Each circle (D=div, C=cmp, S=sub, A=add) represents a call to a function in the page that holds these arithmetic functions. We count the exact number of instructions between two consecutive invocations that hit this page. The instruction counts reveal branch outcomes.

With a trace including the weights of instruction counts collected between two consecutive accesses to the page that holds the arithmetic operations (A, C, D, S), we apply a set of divide-and-conquer rules to reconstruct the control flow for `fp_invmod_slow`.

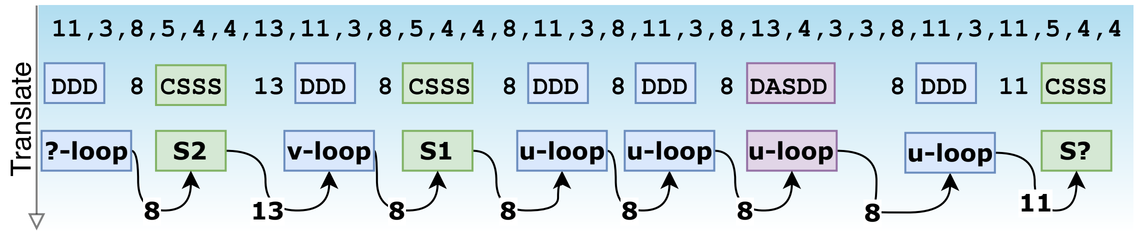


Figure 6.24: An example cut of a trace that is recovered from `fp_invmod_slow`. First, we replace the weights according to Rules 1, 2, and 3. Then we use other transitions (Rules 4 and 5) to recover the whole control flow sequence.

These rules start by translating the recovered weights to corresponding generic blocks. For example, every time the algorithm executes an iteration of a division loop (u/v-loop), we observe either the sequence $D \rightarrow D \rightarrow D$, or the sequence $D \rightarrow A \rightarrow S \rightarrow D \rightarrow D$. Each of these sequences generates a consistent set of weights. Similarly, *S1* or *S2* always generates a sequence like $C \rightarrow S \rightarrow S \rightarrow S$. After translating these generic blocks, we can use the remaining transitions to distinguish the exact blocks, *i.e.*, we can recover whether a *S1* or *S2* followed by a set of division loops is equal to a transition from *S1* to *u-loop* or transition from *S2* to *v-loop*. These rules are summarized as follow:

- **Rule 1:** $? \xrightarrow{11} ? \xrightarrow{3} ? = D \rightarrow D \rightarrow D$.
- **Rule 2:** $? \xrightarrow{13} ? \xrightarrow{4} ? \xrightarrow{3} ? \xrightarrow{3} ? = D \rightarrow A \rightarrow S \rightarrow D \rightarrow D$.
- **Rule 3:** $? \xrightarrow{5} ? \xrightarrow{4} ? \xrightarrow{4} ? = C \rightarrow S \rightarrow S \rightarrow S$.
- **Rule 4:** $S? \xrightarrow{13} ? = S2 \rightarrow$ *v-loop*.
- **Rule 5:** $S? \xrightarrow{8} ? = S1 \rightarrow$ *u-loop*.

We first replace some weights according to Rules 1, 2, and 3, which identify if we are in a division loop (u-loop or v-loop) or a comparison and substitution block (S?). Based on the other transitions (Rule 4 and 5), we can determine which state of the comparison and substitution block we have moved from and which division loop we have moved to within the trace. An example sequence from the execution of *fp_invmod_slow* and its translation to the control flow transitions is given in Figure 6.24.

For the compact implementation in *fp_invmod*, we apply the same approach. Figure 6.25 shows the control flow for this implementation after runtime analysis using CopyCat. Similarly, we define a set of rules to translate the trace of instruction counts

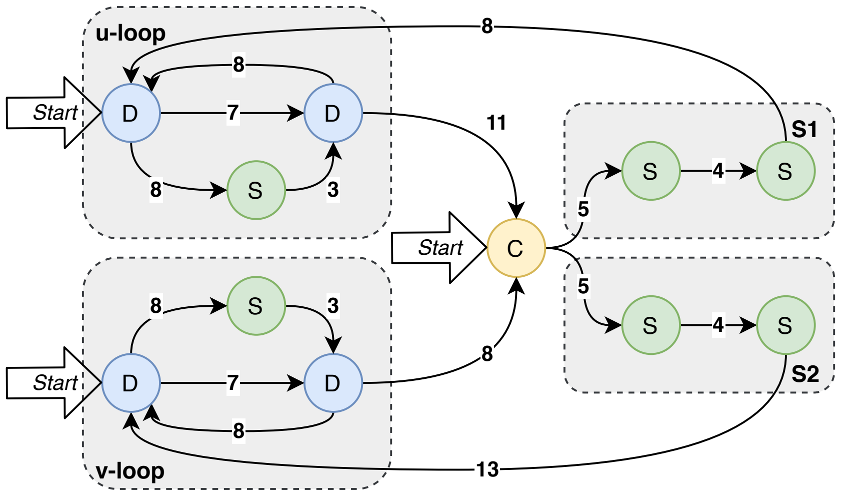


Figure 6.25: Control flow of BEEA in `fp_invmod`.

to control flow transfers of BEEA. Based on Figure 6.25, we modify the first three rules as follows to support control-flow recovery based on the same approach:[5]

- **Rule 1:** $? \xrightarrow{7} ? = D \rightarrow D$.
- **Rule 2:** $? \xrightarrow{8} ? \xrightarrow{3} ? \xrightarrow{3} ? = D \rightarrow S \rightarrow D$.
- **Rule 3:** $? \xrightarrow{5} ? \xrightarrow{4} ? = C \rightarrow S \rightarrow S$.

6.4.2 Single-Trace Attack on DSA Signing

In contrast to previous attacks on BEEA [113] that leak partial information about the nonce, CopyCat recovers the complete control flow from the execution of this implementation with 100 percent precision virtually. As a result, we can perform a single-trace attack on the DSA signing operation. In Section 6.4.4, we generalize this attack and expose multiple vulnerabilities in the *Libgcrypt* library.

DSA key recovery. WolfSSL uses `fp_invmod` to compute the modular inversion of $k_{inv} = k^{-1} \bmod n$, where n is an odd prime. Since we can recover the exact control flow of this computation and the modulus n is public, we step through the execution trace of Algorithm 3, applying each step of the computation according to the recovered path to compute k_{inv} bit by bit. After recovering k_{inv}, recovering the full nonce and private key is trivial: $k = k_{inv}^{-1} \bmod n$, $x = r^{-1}(sk - h) \bmod n$.

[5]Rule 4 and 5 remain the same.

Evaluation. To attack 160-bit DSA, we used a combination of pages in a page-level controlled-channel attack to first reach the beginning of the modular inversion operation for DSA. Then we start CopyCat over the code page for `fp_invmod`. We executed this attack for 100 different signing operations. On average, this attack issues 22,000 IRQs and takes 75 ms to iterate over an average of 6,320 steps for each signature generation. Out of 100 experiments, our single-trace attack successfully recovered the full control flow and the key using the algorithm above, implying that CopyCat reliably reconstructs the entire execution flow. As a result, we can execute a single-trace attack on DSA without the need for multiple signatures.

6.4.3 Single-Trace Attacks on RSA Key Generation

During RSA key generation, WolfSSL checks if a potential prime p is coprime with e by checking if $gcd(e, p-1)$ is equal to 1. This step uses the textbook greatest common divisor (GCD) algorithm, which performs a series of divisions. This algorithm appears to be less vulnerable to control-flow-based key recovery. However, in a later stage, WolfSSL computes $d = e^{-1} \bmod \lambda(N)$ and the CRT parameter $q^{-1} \bmod p$ using the BEEA. WolfSSL always generates the CRT parameters during RSA key generation.[6]

Key recovery from a $q^{-1} \bmod p$ trace. Compared to $k^{-1} \bmod n$, this attack is more challenging since, in this case, both operands p and q are unknown. We give a novel and efficient attack that recovers the secret RSA parameters p and q using CopyCat. We use the relationship of the public modulus $N = pq$ and the execution trace of the BEEA on $q^{-1} \bmod p$, which provides enough information to recover the factorization of N. The main idea is that the BEEA algorithm works sequentially from the least significant bits of p and q. Thus if we iteratively guess bits of p and q starting from the least significant bits, we can verify that a guess matches the relevant steps of the BEEA execution trace, as well as the constraint that $N = pq$ for the bits guessed so far, and eliminate guesses that do not. This algorithm resembles the branch-and-prune algorithm of [145], with new constraints.

We propose Algorithm 4 to recover p and q using only knowledge of N and the execution trace of the BEEA on $q^{-1} \bmod p$. The algorithm starts by initializing a list of hypotheses for values of the least significant bits of q and p. Each hypothesis keeps track of the current step, bit position b, and the hypothesized values of $p_s = p \bmod 2^b$ and $q_s = q \bmod 2^b$. Among the four possible assignments for the $(b+1)^{st}$ bits of p and q in Step 7, there will be two choices satisfying the constraint that $pq \equiv N \bmod 2^{b+1}$. We

[6]`wc_MakeRsaKey` at https://github.com/wolfSSL/wolfssl/blob/48c4b2fedc/wolfcrypt/src/rsa.c#L3726 invokes BEEA multiple times during RSA Key generation.

evaluate the BEEA algorithm for these new guesses up to the number of bits guessed so far and check this deterministic algorithm evaluation on the guess against the ground truth execution trace t. We then do a depth-first search prioritized by the number of steps in which the algorithm is executed correctly and terminate when we have found a candidate for which $pq = N$ holds.

Evaluation. We executed an attack similar to Section 6.4.2 to collect traces from the modular inversion of $q^{-1} \bmod p$, as it is computed by `fp_invmod_slow`. We tried this attack on 100 different 2048-bit RSA key generations. On average, we iterate over 39,400 steps by issuing 106490 IRQs in 365 ms. However, the average time to collect a trace can take up to a second, depending on the prime number generation's execution time. The attack takes 20 seconds to recover the key from a trace. All 100 trials of the attack successfully recovered the keys.

Key recovery from an $e^{-1} \bmod \lambda(N)$ trace. In contrast to previous attacks on this computation [58], we propose a different algorithmic attack that takes advantage of the fact that CopyCat can recover the entire control flow of this algorithm. As a result, one can carry out a single-trace attack for any value of e, both large or small. This observation shows that the proposed masking countermeasure in [58] is insecure against our strong CopyCat adversary.

Our goal is to recover the RSA primes p and q using the trace of the BEEA for $d = e^{-1} \bmod \lambda(N)$. The modulus N and the public exponent e are known, while $\lambda(N)$ is secret. We present a modified branch-and-prune technique in Algorithm 5 that recovers the factors p and q for a large fraction of generated RSA keys.

The main idea is to iteratively guess bits of p and q starting from the least significant bits, then verify that $pq = N$ and the relevant steps of the BEEA execution trace match the guess so far. However, the BEEA is computed on e and $\lambda(N) = (p-1)(q-1)/\gcd(p-1, q-1)$. We do not know $\gcd(p-1, q-1)$ and must guess it for this algorithm, but with high probability, it only has small factors and can be brute-forced. For simplicity, we specialize to the case of $\gcd(p-1, q-1) = 2^i$ for small integer i below, but the analysis can be extended to other candidate small primes with more brute force effort.

For each guess 2^i for $\gcd(p-1, q-1)$, we iteratively generate guesses for p_s and q_s, compute $\phi_s = (p_s - 1)(q_s - 1)$ and then $\lambda_s = \phi_s/2^i$. We compare the execution trace t to the execution trace for λ_s and e. The algorithm either returns p and q or it fails to recover p and q if $\phi/\lambda(N) \neq 2^i$.

Analysis. The algorithm will succeed whenever $\phi/\lambda = 2^i$ for small i. For non-powers of 2, the test against the BEEA execution trace in Step 15 will likely fail and cause this branch to be pruned. Since $p = 2p' + 1$ and $q = 2q' + 1$ for some $p', q' \in \mathbb{Z}$, we have $\lambda(N) = \mathrm{lcm}(p-1, q-1) = 2\mathrm{lcm}(p', q')$. From the prime number theorem [305], the probability that two random integers are coprime converges to $\prod_{p \in primes}(1 - 1/p^2) = \frac{6}{\pi^2} \approx 61\%$ as the size of the integers increases. In other words, if we run Algorithm 5 for only $i = 1$, it will succeed 61% of the time when p' and q' are actually coprime. If we allow p' and q' to have even factors we obtain a probability of $\prod_{p \in primes, p>2}(1 - 1/p^2) = \frac{8}{\pi^2} \approx 81\%$. This analysis means that even for a modest number of iterations, e.g., $\ell = 8$, we have nearly 81% success probability. Our experiments confirm these estimates.

Evaluation. We tried this attack on 100 different key generation efforts (2048-bit key). On average, we iterate over 81,090 steps by issuing 230,050 interrupts per attack in 800ms. The average time to collect a trace is about a second, and the attack takes about 30 seconds to successfully recover the key for 81% of the keys when $\mathrm{lcm}((p-1)(q-1)) \equiv (p-1)(q-1)/2^i$.

Revisiting masking protection. Earlier attacks required bruteforcing the value of e [365]. Our algorithm works for arbitrary, even for the maximum possible length of e. Thus increasing the size of e by choosing a larger public exponent or masking is insufficient to mitigate our attack. Aldaya et al. [17] proposed masking e by computing $b = (er)^{-1} \bmod \lambda(N)$ for a random r such that $\gcd(r, \lambda(N)) = 1$. The private key then can be computed as $d = rb \bmod \lambda(N)$. In this proposal, they have even suggested that r can be hardcoded. We tested our attack for a hardcoded (known) choice of r and verified that key recovery works in this case. Alternatively, if r is not hardcoded but we have a trace for the initial $\gcd(r, \lambda(N))$ computation using binary gcd, we can again decode it (with the knowledge of N) to recover r. With r recovered, the attack proceeds as before, *i.e.*, from the execution trace of $b = (er)^{-1} \bmod \lambda(N)$ we recover p and q by running Algorithm 5 with er supplied as input instead of e. Since Algorithm 5 is agnostic with respect to the size of e, it will handle the full size er and recover p and q.

6.4.4 CopyCat-Based Cryptanalysis

Now that we have empirically verified through real-world attacks that CopyCat can recover the runtime control flow of all the branches deterministically, we analyze similar cryptographic implementations in other open-source libraries. This analysis covers the latest Libgcrypt 1.8.5, OpenSSL 1.1.1d, and Intel IPP Crypto [158]. OpenSSL and Intel IPP Crypto are particularly crucial for products using Intel SGX. Intel has an official wrapper around OpenSSL, called Intel SGX-SSL [161]. The current version of Intel SGX-SSL pulls the stable OpenSSL 1.1.1d. Intel IPP Crypto is the official cryptographic library by Intel, and many Intel products, including Intel SGX SDK [174], are using it. Section 6.4.4 summarizes our findings in this paper regarding vulnerable code paths.

	Operation (Subroutine)	Implementation	Secret Branch	Exploitable Computation → Vulnerable Callers		Single-Trace Attack
WolfSSL	Scalar Multiply (`wc_ecc_mulmod_ex`)	Montgomery Ladder w/ Branches	✓	✓	$(k \times G) \rightarrow$ `wc_ecc_sign_hash`	✗
	Greatest Common Divisor (`fp_gcd`)	Euclidean (Divisions)	✓	✗	N/A	N/A
	Modular Inverse (`fp_invmod`)	BEEA	✓	✓	$(k^{-1} \bmod n) \rightarrow$ `wc_DsaSign`	✓
					$(q^{-1} \bmod p) \rightarrow$ `wc_MakeRsaKey`	✓
					$(e^{-1} \bmod \Lambda(N)) \rightarrow$ `wc_MakeRsaKey`	✓
Libgcrypt	Greatest Common Divisor (`mpi_gcd`)	Euclidean (Divisions)	✓	✗	N/A	N/A
	Modular Inverse (`mpi_invm`)	Modified BEEA [205, Vol II, §4.5.2]	✓	✓	$(k^{-1} \bmod n) \rightarrow$ `{dsa,elgamal}.c::sign,_gcry_ecc_ecdsa_sign`	✓
					$(q^{-1} \bmod p) \rightarrow$ `generate_{std,fips,x931}`	✓
					$(e^{-1} \bmod \Lambda(N)) \rightarrow$ `generate_{std,fips,x931}`	✓
OpenSSL	Greatest Common Divisor (`BN_gcd`)	BEEA	✓	✓	$gcd(q-1, p-1) \rightarrow$ `RSA_X931_derive_ex`	✓
	Modular Inverse (`BN_mod_inverse_no_branch`)	BEEA w/ Branches	✗	N/A	N/A	N/A
IPP Crypto	Greatest Common Divisor (`ippsGcd_BN`)	Modified Lehmer's GCD	✓	?	$gcd(q-1, e) \rightarrow$ `cpIsCoPrime`	N/A
				?	$gcd(p-1, q-1) \rightarrow$ `isValidPriv1_rsa`	N/A
	Modular Inverse (`cpModInv_BNU`)	Euclidean (Divisions)	✓	✗	N/A	N/A

Table 6.6: An overview of applicability of CopyCat on cryptographic libraries: WolfSSL, Libgcrypt, OpenSSL, IPP Crypto.

Libgcrypt analysis. Libgcrypt uses a custom implementation of the extended Euclidean algorithm to compute modular inverses (Algorithm 6). This algorithm is derived from an exercise from The Art of Computer Programming [205, Vol II, §4.5.2, Alg X]. The algorithm is an adaptation of Algorithm X to use the efficient divide by two reduction steps in the binary Euclidean algorithm. The algorithm computes a vector (u_1, u_2, u_3) such that $uu_1 + vu_2 = u_3 = \gcd(u, v)$ using auxiliary vectors $(v_1, v_2, v_3), (t_1, t_2, t_3)$. The iterations preserve the invariants $ut_1 + vt_2 = t_3$, $uu_1 + vu_2 = u_3$ and $uv_1 + vv_2 = v_3$. This algorithm is used in numerous places for secret operations.

$k^{-1} \bmod n$ in DSA, ECDSA and ElGamal: The DSA, ECDSA and ElGamal signature schemes all require computing $k^{-1} \bmod n$. Libgcrypt computers all of these use cases using Algorithm 6. We derive a single-trace attack similar to Section 6.4.2 that recovers all the algorithm branches during this computation. This technique trivially leaks k^{-1} for each of these algorithms in a single-trace attack. As a result, they are all vulnerable to the attack described in Section 6.4.2. Note that DSA and ElGamal do not use any masking countermeasure, and we discuss below how the masking countermeasure for ECDSA is insecure.

ECDSA masking countermeasure: We identified two vulnerabilities related to masking countermeasure for ECDSA signing in Libgcrypt, as shown in Listing 8, which leaves it vulnerable to attacks against Algorithm 6 and a single-trace attack during the computation of $k^{-1} \bmod n$. Using a randomly chosen blinding variable b, Libgcrypt computes the blinded signature as $s_b = k^{-1}(hb + bdr) \bmod n$. To compute the unblinded signature, it computes $s = s_b b^{-1} \bmod n$. The first vulnerability is that $k^{-1} \bmod n$ is not blinded, so a single-trace attack on this operation simply recovers the nonce k. This blinding should be modified to $s_b = (kb)^{-1}(h + xr) \bmod n$, and this can be unblinded by computing $s = s_b b \bmod n$.

The second vulnerability is that since, in this blinding scheme, the implementation needs to invert b, Libgcrypt computes the $b^{-1} \bmod n$ using the same vulnerable technique (Listing 9). Therefore, a single-trace attack can also recover the blinding value.

RSA input masking: To avoid timing attacks, RSA decryption and signing in Libgcrypt use masking on the input ciphertext or message. For a random variable r and input ciphertext c, the decryption is performed on $m_b = (cr^e)^d \bmod n = c^d r \bmod n$. The message can then be unblinded by computing $m = m_b r^{-1} = c^d \bmod n$. Unfortunately, the vulnerable modular inverse function is responsible for computing $r^{-1} \bmod n$. As a result, a single-trace attack can recover the blinding factor, rendering this countermeasure ineffective.

RSA key generation: Libgcrypt has three RSA key generation subroutines for different use cases: `generate_std`, `generate_fips` and `generate_x931` all use the vulnerable `mpi_invm` function to compute both $q^{-1} \bmod p$ and $e^{-1} \bmod \lambda(N)$, and are vulnerable to the attacks described in Section 6.4.3.

OpenSSL analysis. After several iterations [113, 365], OpenSSL implemented a constant-time modular inversion function, `BN_mod_inverse_no_branch` for DSA, ECDSA, and RSA key generation. Various critical primitives use this function to compute the GCD. However the legacy binary GCD function is still supported in the latest OpenSSL code base, version 1.1.1d, in the function `BN_gcd` (cf. Appendix Algorithm 7). The subroutine `RSA_X931_derive_ex`, which is responsible for generating RSA keys according to the X.931 standard, uses this function during the computation of $\lambda(N) = \mathrm{lcm}(p-1, q-1) = (p-1)(q-1)/gcd(p-1, q-1)$, as shown in Listing 10. Thus we can apply our attack technique from Section 6.4.3 to recover the RSA private key from the computation of $gcd(p-1, q-1)$.

Analysis of Intel IPP Crypto. The *Intel IPP Crypto* library uses a conventional Euclidean algorithm to compute modular inverses. This algorithm performs a series of division operations in a loop. While CopyCat can recover the precise number of division operations, this leakage does not seem to be exploitable during the RSA key generation [248, §6].

On the other hand, for computing the GCD, Intel IPP Crypto uses a modified version of Lehmer's GCD algorithm [318]. Lehmer's GCD algorithm and Intel's modified implementation are not constant time and have secret-dependent branches [160]. This GCD implementation is only used during RSA key generation, where only a single-trace attack results in a vulnerability. Our analysis in Section 6.4.3 does not directly apply to this algorithm, and we leave the study and potential exploitability of this implementation for future work. This potential oversight in Intel's GCD implementation once more illustrates the intricacies of applying Intel's own recommended constant-time programming guidelines [180].

More single-Trace attack evaluations. We replicated the attack in Section 6.4.2 using synthetic traces from Algorithm 6. We ran the attack on 100 different $k^{-1} \bmod n$ and recovered k_{inv} and the secret key in all cases. The attack applies to ElGamal as well by computing the private key $x = r^{-1}(h - sk) \bmod (p - 1)$.

Single-Trace attack on RSA Key Generation (Libgcrypt, OpenSSL). We replicated synthetic traces of branches from OpenSSL's binary GCD algorithm executed on $\gcd(q - 1, p - 1)$. We applied Algorithm 4 with a modified test function modeling this algorithm and applied a heuristic to match the appropriate number of trace steps to the bits guessed so far. We ran the attack using synthetic traces for 100 different 256-bit RSA keys. We chose this key size to verify the correctness of our algorithm efficiently. Our attack successfully recovered every key. We similarly replicated the same attack as Section 6.4.3 with a test function following Algorithm 6. Similarly, we ran the attack using synthetic traces for 100 different 256-bit RSA keys, and the attack was successful in all cases.

Listing 4 `fp_invmod_slow` implements modular inversion using the binary extended Euclidean algorithm (BEEA). The subroutines `fp_iseven` and `fp_isodd` are inlined and simply check the last bit of their operand. The arithmetic subroutines: `fp_sub`, `fp_div_2`, `fp_cmp` can all fit within the same page.

```
static int fp_invmod_slow (fp_int * a, fp_int * b, fp_int * c){...
top:
 while (fp_iseven (u) == FP_YES) { /* 4.  while u is even do */
   fp_div_2 (u, u);               /* 4.1 u = u/2 */
   if (fp_isodd (A) == FP_YES ||   /* 4.2 if A or B is odd then */
   fp_isodd (B) == FP_YES) {
     fp_add (A, y, A);             /* A = (A+y)/2 */
     fp_sub (B, x, B);             /* B = (B−x)/2 */
   }
   fp_div_2 (A, A);                /* A = A/2 */
   fp_div_2 (B, B);                /* B = B/2 */
 }
 while (fp_iseven (v) == FP_YES) { /* 5.  while v is even do */
   fp_div_2 (v, v);               /* 5.1 v = v/2 */
   if (fp_isodd (C) == FP_YES ||   /* 5.2 if C or D is odd then */
   fp_isodd (D) == FP_YES) {
     fp_add (C, y, C);             /* C = (C+y)/2 */
     fp_sub (D, x, D);             /* D = (D−x)/2 */
   }
   fp_div_2 (C, C);                /* C = C/2 */
   fp_div_2 (D, D);                /* D = D/2 */
 }
 if (fp_cmp (u, v) != FP_LT) {     /* 6.  if u >= v then */
   fp_sub (u, v, u);              /* u = u − v */
   fp_sub (A, C, A);               /* A = A − C */
   fp_sub (B, D, B);               /* B = B − D */
 } else {
   fp_sub (v, u, v);              /* v − v − u */
   fp_sub (C, A, C);               /* C = C − A */
   fp_sub (D, B, D);               /* D = D − B */
 }
 if (fp_iszero (u) == FP_NO) goto top;     /* if not zero goto step 4 */
```

Listing 5 `fp_invmod` implements modular inversion using the binary extended Euclidean algorithm (BEEA). This function is similar to Listing 4, but only supports odd numbers as the second parameter.

```
/* c = 1/a (mod b) for odd b only */
int fp_invmod(fp_int *a, fp_int *b, fp_int *c) {...
top:
  while (fp_iseven (u) == FP_YES){  /* 4.  while u is even do */
    fp_div_2 (u, u);                /* 4.1 u = u/2 */
    if (fp_isodd (B) == FP_YES){    /* 4.2 if B is odd then */
      fp_sub (B, x, B);
    }
    fp_div_2 (B, B);                /* B = B/2 */
  }
  while (fp_iseven (v) == FP_YES){  /* 5.  while v is even do */
    fp_div_2 (v, v);                /* 5.1 v = v/2 */
    if (fp_isodd (D) == FP_YES){    /* 5.2 if D is odd then */
      fp_sub (D, x, D);             /* D = (D-x)/2 */
    }
    fp_div_2 (D, D);                /* D = D/2 */
  }
  if (fp_cmp (u, v) != FP_LT){      /* 6.  if u >= v then */
    fp_sub (u, v, u);               /* u = u - v */
    fp_sub (B, D, B);               /* B = B - D */
  } else {
    fp_sub (v, u, v);               /* v - v - u */
    fp_sub (D, B, D);               /* D = D - B */
  }
  if (fp_iszero (u) == FP_NO) goto top;    /* if not zero goto step 4 */
```

Listing 6 `wc_MakeRsaKey`.

```
if (err == MP_OKAY)           /* key->d = 1/e mod lcm(p-1, q-1) */
    err = mp_invmod(&key->e, &tmp3, &key->d);
if (err == MP_OKAY)           /* key->n = pq */
    err = mp_mul(&p, &q, &key->n);
if (err == MP_OKAY)           /* key->dP = d mod(p-1) */
    err = mp_mod(&key->d, &tmp1, &key->dP);
if (err == MP_OKAY)           /* key->dQ = d mod(q-1) */
    err = mp_mod(&key->d, &tmp2, &key->dQ);
if (err == MP_OKAY)           /* key->u = 1/q mod p */
    err = mp_invmod(&q, &p, &key->u);
```

Algorithm 3 Modular inversion using the BEEA. In the optimized compact implementation when the modulus is odd, highlighted (blue) statements are removed.

```
1: procedure MODINV($u$, modulus $v$)
2:     $b_i \leftarrow 0$ $d_i \leftarrow 1, u_i \leftarrow u, v_i = v$, $a_i \leftarrow 1$, $c_i \leftarrow 0$
3:     while $isEven(u_i)$ do
4:         $u_i \leftarrow u_i/2$
5:         if $isOdd(b_i)$ then
6:             $b_i \leftarrow b_i - u$, $a_i \leftarrow a_i + v$
7:         $b_i \leftarrow b_i/2$, $a_i \leftarrow a_i/2$
8:     while $isEven(v_i)$ do
9:         $v_i \leftarrow v_i/2$
10:        if $isOdd(d_i)$ then
11:            $d_i \leftarrow d_i - u$, $c_i \leftarrow c_i + v$
12:        $d_i \leftarrow d_i/2$, $c_i \leftarrow c_i/2$
13:    if $u_i > v_i$ then
14:        $u_i \leftarrow u_i - v_i$, $b_i \leftarrow b_i - d_i$, $a_i \leftarrow a_i - c_i$
15:    else
16:        $v_i \leftarrow v_i - u_i$, $d_i \leftarrow d_i - b_i$, $c_i \leftarrow c_i - a_i$
17:
       return $d_i$
```

Algorithm 4 Recovering p and q from trace of $q^{-1} \bmod p$.

```
1: procedure RECOVER_PQ(trace $t$, modulus $N$)
2:     $h \leftarrow (-test\_t(t, 1, 1), 1, 1, 1)$
3:     while $h$ do
4:         $steps, b, p, q \leftarrow hpop(h)$
5:         if $p.q = N$ then
6:             return $p, q$
7:         $g \leftarrow (p, q), (p + 2^b, q), (p, q + 2^b), (p + 2^b, q + 2^b)$
8:         for $p_s, q_s$ in $g$ do
9:             if $mod(p_s.q_s, 2^{b+1}) = mod(N, 2^{b+1})$ then
10:                $hpush(h, (-test\_t(trace, p_s, q_s), b + 1, p_s, q_s))$
```

Algorithm 5 Recovering p and q from trace of $e^{-1} \bmod \lambda$.

```
1: procedure RECOVER_PQ(trace $t$, $e$, modulus $N$)
2:     $h \leftarrow (-test\_t(t, 0, e), 1, 1, 1)$
3:     while $h$ do
4:         $steps, b, p, q \leftarrow hpop(h)$
5:         if $p.q = N$ then return $p, q$
6:         $g \leftarrow (p, q), (p + 2^b, q), (p, q + 2^b), (p + 2^b, q + 2^b)$
7:         for $p_s, q_s$ in $g$ do
8:             if $mod(p_s.q_s, 2^{b+1}) = mod(N, 2^{b+1})$ then
9:                 $\phi = (p_s - 1)(q_s - 1)$
10:                for $i = 1, \ldots, 2^\ell$ do
11:                    if $p_s q_s > N$ or $mod(\phi, 2^i) \neq 0$ then
12:                        continue
13:                    $\lambda = \phi/2^i$
14:                    $newsteps = test\_t\_lamda(t, \lambda, e)$
15:                    if $newsteps >= b + 1$ then:
16:                        $hpush(h, (-newsteps, b + 1, p_s, q_s))$
       return fail
```

Listing 7 `wc_ecc_mulmod_ex` implements scalar multiplication using a bit-by-bit always-add-and-double algorithm. The function protects against both timing and cache attacks by executing dummy instructions. For brevity, error checking and code sections that are not relevant to our attack are removed.

```
int wc_ecc_mulmod_ex(mp_int* k, ecc_point *G, ecc_point *R, mp_int* a, mp_int* modulus,
      int map, void* heap) { ...
for (;;) {
if (--bitcnt == 0) { /* grab next digit as required */
  if (digidx == -1) {
      break;
  }
  buf = get_digit(k, digidx);
  bitcnt = (int)DIGIT_BIT;
  --digidx;
}
i = (buf >> (DIGIT_BIT - 1)) & 1;  /* grab the next msb from the multiplicand */
buf <<= 1;
if (mode == 0) {
  mode = i;  /* timing resistant - dummy operations */
  err = ecc_projective_add_point(M[1], M[2], M[2], a, modulus, mp);...
  err = ecc_projective_dbl_point(M[2], M[3], a, modulus, mp);...
}...
err = ecc_projective_add_point(M[0], M[1], M[i^1], a, modulus, mp);...
err = ecc_projective_dbl_point(M[2], M[2], a, modulus, mp);...
} /* end for */...}
```

Algorithm 6 Modular inversion using a variant of BEEA.

```
1: procedure ModInv($u$, modulus $v$)
2:     $u_1 \leftarrow 1, u_2 \leftarrow 0, u_3 \leftarrow u$
3:     $v_1 \leftarrow v, v_2 \leftarrow u_1 - u, v_3 \leftarrow v$
4:     if $isOdd(u)$ then
5:         $t_1 \leftarrow 0, t_2 \leftarrow -1, t_3 \leftarrow -v$
6:     else
7:         $t_1 \leftarrow 1, t_2 \leftarrow 0, t_3 \leftarrow u$
8:     while $t_3 \neq 0$ do
9:         while $isEven(t_3)$ do
10:            if $isOdd(t_1)$ or $isOdd(t_2)$ then
11:                $t_1 \leftarrow t_1 + v, t_2 \leftarrow t_2 - u$
12:            $t_1 \leftarrow t_1/2, t_2 \leftarrow t_2/2, t_3 \leftarrow t_3/2$
13:        if $t_3 > 0$ then
14:            $u_1 \leftarrow t_1, u_2 \leftarrow t_2, u_3 \leftarrow t_3$
15:        else
16:            $v_1 \leftarrow v - t_1, v_2 \leftarrow -u - t_2, v_3 \leftarrow -t_3$
17:        $t_1 \leftarrow u_1 - v_1, t_2 \leftarrow u_2 - v_2, t_3 \leftarrow u_3 - v_3$
18:        if $t_1 < 0$ then
19:            $t_1 \leftarrow t_1 + v, t_2 \leftarrow t_2 - u$
       return $u_1$
```

Listing 8 The masking protection for ECDSA leaves the $k^{-1} \bmod n$ operation vulnerable to our single-trace attack.

```
mpi_mulm (dr, b, skey->d, skey->E.n);
mpi_mulm (dr, dr, r, skey->E.n);     /* dr = d*r mod n (blinded) */
mpi_mulm (sum, b, hash, skey->E.n);
mpi_addm (sum, sum, dr, skey->E.n);  /* sum = hash + (d*r) mod n  (blinded) */
mpi_mulm (sum, bi, sum, skey->E.n);  /* undo blinding by b^-1 */
mpi_invm (k_1, k, skey->E.n);        /* k_1 = k^(-1) mod n  */
mpi_mulm (s, k_1, sum, skey->E.n);   /* s = k^(-1)*(hash+(d*r)) mod n */
```

Listing 9 `_gcry_ecc_ecdsa_sign` computes the modular inverse of the blinding factor b using a vulnerable function.

```
do {    _gcry_mpi_randomize (b, qbits, GCRY_WEAK_RANDOM);
        mpi_mod (b, b, skey->E.n);
} while (!mpi_invm (bi, b, skey->E.n));
```

Listing 10 `RSA_X931_derive_ex` uses `BN_gcd` to compute $\lambda(N)$, exposing p and q to our attack.

```
if (!BN_sub(r1, rsa->p, BN_value_one())) goto err;     /* p-1 */
if (!BN_sub(r2, rsa->q, BN_value_one())) goto err;     /* q-1 */
if (!BN_mul(r0, r1, r2, ctx)) goto err;                /* (p-1)(q-1) */
if (!BN_gcd(r3, r1, r2, ctx)) goto err;
```

Algorithm 7 OpenSSL Binary GCD Algorithm.

```
1:  procedure GCD($a$, $b$)
2:      $s \leftarrow 0$
3:      if $a < b$ then
4:          $a, b \leftarrow b, a$
5:      while $b \neq 0$ do
6:          if $isOdd(a)$ then
7:              if $isOdd(b)$ then                          ▷ a is odd, b is odd
8:                  $a \leftarrow a - b, a \leftarrow a/2$
9:                  if $a < b$ then
10:                     $a, b \leftarrow b, a$
11:             else                                        ▷ a is odd, b is even
12:                 $b \leftarrow b/2$
13:                 if $a < b$ then
14:                     $a, b \leftarrow b, a$
15:         else
16:             if $isOdd(b)$ then                          ▷ a is even, b is odd
17:                 $a \leftarrow a/2$
18:                 if $a < b$ then
19:                     $a, b \leftarrow b, a$
20:             else                                        ▷ a is even, b is even
21:                 $a \leftarrow a/2, b \leftarrow b/2, s \leftarrow s + 1$
22:     if $s > 0$ then
23:         $a \leftarrow a * (2^s)$
24:     return $a$
```

Chapter 7

Revisiting Isolated and Trusted Execution

In this final chapter, we first discuss various approaches to mitigate the discussed vulnerabilities on existing and future system (§7.1). Next, in Section 7.2, we discuss open challenges and future opportunities for microarchitectural security. Finally, we conclude our findings while providing an assessment check-list for microarchitectural cryptanalysis in Section 7.3.

7.1 Countermeasure Discussions

This section discuses three different path in countering microarchitectural vulnerabilities based on attack detection (§7.1.1), software hardening (§7.1.2), and architectural mitigations and fixes (§7.1.3). Finally, we report the coordinated disclosure process with vendors regarding our findings (§7.1.4).

7.1.1 Attack Detection

Detect vulnerabilities when attackers are exploiting them at runtime is known as a cost-effective approach for risk mitigation. Traditional intrusion detecting systems (IDS) and antivirus software are typical examples of attack detection [86]. Generally, the detection system alerts a high-privilege entity to respond to an alleged attack e.g., kill the compromised process, shut down the machine, or report to an administrator. For example, an IDS tailored for inspecting API calls and network traffic originating from a TPM may detect attacks like the TPM-FAIL conducted by network and user adversaries. However, such detection techniques do not scale to all threat models like the system-level adversaries. More importantly, IDS rules suffer from false positives and can be avoided in most cases by determined adversaries; an adversary can introduce a random delay between requests or combine the malicious requests with benign ones to circumvent detection.

Microarchitectural attack detection. Researchers have also proposed detection techniques for microarchitectural attacks, e.g., to detect cache attacks in cloud environments [52, 74, 389]. The hope is that developers can use dynamic tools to detect attacks for microarchitectural vulnerabilities that cannot be mitigated at the system and architecture level. Some of the proposed methods to detect cache attacks at runtime utilize hardware performance counters [52, 389] and transactional synchronization extensions (TSX) [72, 311] to detect abnormal microarchitectural behavior. Proposals based on performance counters that monitor cache activities such as the number of cache misses are incapable of detecting MemJam or Spoiler, as the attack does not introduce irregular cache activities. Although one might argue using other performance counters for detection, e.g., two of the counters `Ld_Blocks_Partial:Address_Alias` and `Cycle_Activity:Stalls_Ldm_Pending` have high correlations with Spoiler, its practicality is debatable.

To engineer a microarchitectural detection system, a monitoring agent needs to occupy an active thread and actively evaluate the number of memory stalls. In our experiments for the MemJam case, performing 50 million observations takes less than a minute. If such a detector exists, it has to monitor with a higher frequency than the attack; otherwise, attackers will outperform it before detecting suspicious behavior is possible. Using TSX as a detector with low false-positives would also not be practical since the read-after-write hazards are common phenomena, and TSX could fail due to other issues [163], resulting in a lot of false positives.

Detecting interrupts. Similarly, in the SGX world, detection of unexpected interrupts, as proposed in the literature [72], may be applicable to mitigate CopyCat. CopyCat relies on the ability to single-step enclaved execution, which is within Intel SGX's threat model [164, 351]. While SGX enclaves remain explicitly interrupt-unaware by design, some research proposals [72, 311] retrofit hardware support for transactional memory to detect suspicious interrupt rates as a side-effect of an ongoing attack. However, such features are not commonly available on off-the-shelf SGX hardware, and they would not fundamentally address the attack surface as CopyCat adversaries are likely to develop stealthier attack techniques [353, 360] that remain under the radar of heuristic defenses.

Limiting factors for adoption. Generally speaking, detection techniques even for well-understood attack vectors prone to false positives and false negatives. False positives would essentially make a detection system undesirable since it reduces the quality of service and stability. False negatives are cases when the detection system has failed to detect an attack. In the context of microarchitectural attacks, detection techniques additionally suffer from the following limitations:

1. **Invisibilty:** Commodity microarchitectures do not have dedicated sensors for monitoring security-critical events. Previous work exploits CPU features not designed for security, which are fundamentally inaccurate and limited in the visibility of what happens inside the processor.

2. **Partial coverage:** Proposed detectors address only an instance of an attack or a small number of attacks; they generalize to new cases of attacks or, worse unknown attack vectors. With this lack of generalization, it is not convincing to deploy these mitigations because one can always use a slightly different technique to compromise the same security guarantees.

3. **Naive attackers:** Previous detection techniques assume that the attackers are naive and always execute the same code sequence to exploit microarchitectural attacks. However, in practice, if an attacker were after executing these attacks, they could make it stealthier by obfuscating their footprint or adding delays, as mentioned earlier.

These problems contribute to microarchitectural attack detection, rarely being considered a practical solution. The lack of visibility is mostly due to the obscurity of processors. If processor manufacturers decide to provide more runtime information or dedicate special sensors, this could improve detection techniques. However, we are unsure if there is enough incentive for chip manufacturers to create more visibility about the processor's operations. Note that adding more visibility is also counter-intuitive to protecting intellectual property, as competitors can use these interfaces to reverse engineer and gain more insights about the underlying design.

In FortuneTeller [135], we have proposed a detection technique based on profiling the expected/benign behavior of applications using neural networks. Although the lack of visibility still limits our strategy, we could still improve over the other two restricting factors by demonstrating better false positive and false negative rates and detecting attacks not observed before. However, we believe that the incentive to use detection systems, in general, even if they function with outstanding performance, will depend on if we see attackers exploiting microarchitectural vulnerabilities.

7.1.2 Hardening Applications

Attacks such a MemJam or CopyCat exploit secret-depend memory operations and branches with a high spatial resolution. These attacks rely on finding such vulnerable patterns in the victim application. However, attacks like the Medusa do not rely on the code pattern or presence of code gadgets in a victim application. Researchers and engineers have pursued several software-based methods to harden applications against microarchitectural attacks when it is possible for the former.

Constant-time programming. Constant-time techniques can prevent some side channels in software and have already seen some adoptions by researchers and practitioners. Tromer et al. proposed several strategies such as alternative lookup tables, data-independent memory access patterns, static or disabled cache, and cache state normalization to defend against cache timing attacks [340]. Some RSA and ECC implementations use the scatter-gather technique to hide the secret-dependent cache access footprint [51]. These techniques generally incur additional development and execution costs; such hardened constant-time implementations are usually designed by experts who know the underlying architecture and side-channel domain. Indeed, the analyzed cryptographic implementations use some measures to prevent well-known leakages, though they have come short of protecting against our discoveries.

In the IPP Crypto library, the `Cipher_DES` and `Safe2Encrypt_RIJ128` achieve consistent cache-access pattern by ensuring that the same cache lines are accessed every time regardless of the processed data. The 4-byte spatial resolution of MemJam, however, thwarts this countermeasure by providing intra-cache-line visibility. Attacks such as MemJam show that uniform cache access pattern, cache state normalization [340] and scatter-gather technique [51] fail to protect cryptographic implementations. One approach to restore security and protect against MemJam is to apply constant memory accesses with a 4-byte granularity. That would require accessing every fourth byte of the table for each memory lookup to maintain a uniform memory footprint. At that point, it is easier to access all entries each time, resting assured that there is no other hidden effect in the microarchitecture resulting in a leak with byte granularity.

Similarly, to mitigate the timing vulnerabilities of TPM-Fail, the standard defense is to deploy constant-time techniques as firmware and software patches or replace the vulnerable TPM when patching is not feasible. Intel has promised patches for Intel fTPM, which Intel Management Engine executes these updated patches. We also provided our tools and techniques to STMicroelectronics and evaluated new versions of their products based on our findings. For a high-precision and deterministic attack like CopyCat, right now, the best practice for cryptographic implementations is to avoid secret-dependent branches and memory lookups altogether. WolfSSL applied such a countermeasure to mitigate our attack on ECDSA. Bernstein and Yang proposed a constant-time GCD algorithm for applications like modular inversion [34]. After our report to OpenSSL and Libgcrypt developers, they have adopted this constant-time algorithm.

Hardware extensions and Bitslicing. For symmetric encryption schemes with specific efficiency requirements, the best approach should rely on a dedicated hardware accelerator e.g., AES-NI, since constant-time software implementations are relatively slow. If possible, such performant yet constant-time instruction set extensions should exclusively be used to protect the targeted implementation efficiently. Another approach that is suitable for some schemes such as DES and AES is to use bitsliced software implementations as they avoid memory-related side channels [29]. For ciphers where hardware support is not available, a true constant-time implementation e.g., based on bit-slicing, seems to be the best, albeit slow, alternative. Intel IPP has different variants optimized for various generations of Intel instruction sets [170], and it features other implementations of AES and SM4 in these variants. As shown, the software-only variant of each of the analyzed ciphers is vulnerable to MemJam.

Algorithm tricks. It is preferable to avoid secret-dependent branches, and memory accesses altogether. Some cryptographic schemes can still benefit from reformulating the algorithm without violating its original input, output, and promised security operation. As mentioned in Section 6.4.4, masking the modular inversion input can mitigate one of our demonstrated attacks if appropriately applied, and the blinding value itself is protected. For example, after our report, WolfSSL applied this solution to mitigate our attack on DSA. Some mathematical operations also have alternative implementations that are easier to implement securely. For example, regarding the attack on $q^{-1} \bmod p$ RSA-CRT key generation, we can use the Fermat's Little Theorem computes $q^{p-2} \bmod p$. As a result, the implementation can avoid modular inversion for this operation. Instead, it relies on a modular exponentiation implementation, which happens to be easier to implement securely. However, the previous chapter's evaluation clearly shows that the instruction-granular page access traces extracted by CopyCat or the intra-cache-line memory access pattern from MemJam are more substantial, e.g., one could leak the masking value. Hence, they can target more vulnerable code patterns than prior attacks. Implementations should avoid secret-dependent code paths altogether.

Compiler-based enforcement. Constant-time implementations are not easy to apply, and this is even more difficult for general-purpose applications [323]. Broadly, researchers have also proposed tools to automate the generation of code lacking software-based leakages for further microarchitectural side-channel leakage. For example, `Raccoon` [277] enforces constant-time control flow but stops at the cache-line granularity and makes use of ORAM, which can be very costly. `Escort` [278] and `EncLang` [315] also transform code to constant-time representation. `EncLang` stops at page-level granularity and requires adoption of a new programming language. Escort is not focused on efficient protection against memory side channels and only focuses on arithmetic operations. While these tools address memory leakages, they would need further fine-tuning to address high-resolution attacks like MemJam with 4-byte and CopyCat with instruction-level spatial granularity. While it is more efficient to execute, limited spatial resolution for a defense approach still leaves the door open to attackers.

In particular, to defend generic secret-processing applications against CopyCat, we encourage future research in improved compile-time hardening techniques that may automatically rewrite conditional branches to ensure a constant interrupt counting pattern, regardless of the executed code. The essential requirement would be to ensure that the adversary observes a secret-independent sequence of pages and always counts the same number of instructions between page transitions. The compiler would also have to be explicitly aware of macro fusion, as explained in Section 4.2.2 when balancing the observed instruction counts. We expect further challenges when dealing with secret-dependent loop bounds. One could potentially combine control-flow balancing with existing solutions for data location randomization to handle data accesses as well [47].

Speculative constant-timeness. Spoiler attack exploits the fact that when there is a `load` instruction after several `store` instructions, the physical address conflict causes a high timing behavior. This phenomena happens because of the speculatively executed `load` before all the `stores` are finished executing. No software mitigation can completely erase this problem. However, inserting store fences between the `loads` and `stores` would block this timing behavior. Another yet less robust approach is to execute other instructions between the `loads` and `stores` to decrease the depth of the attack. Note that defenders can not apply these approaches to the user's code space, *i.e.*, the user can always leak the physical address information to amplify other attacks. However, we can apply them to context switching inside the OS with some performance overhead to avoid cross-domain tracking of memory access patterns. Spoiler and more broadly Spectre attacks [207] highlight how constant-time behavior is more challenging to maintain in the microarchitecture's speculative domain. As a result, the notion of constant-time behavior now has a speculative component in which applications need to make sure speculative access to secrets will not result in visible footprints. In this direction, Cauligi et al. have proposed constant-time foundations for the speculative behavior of programs [66].

Finding leakage. Our findings show the importance of stricter verification of cryptographic implementations, especially in the context of trusted computing. To ensure that code does not feature memory leakage, researchers have proposed analysis tools to verify constant-time properties [19]. `MASCAT` [185] is a static code analysis tool, and `wang2017cached` [359] is a dynamic symbolic execution analyzer to detect cache leakages in software implementations. On the same direction, Langley's `ctgrind` and `ct-verif` [19] propose compiler-level verification techniques. Although engineers may extend these identification techniques to support an intra-cache-line leakage model, there is only one proposal that practically considers this sensitive leakage model [93]. In MicroWalk, we proposed a tool based on dynamic binary instrumentation and mutual information analysis to find software-based leakages in cryptographic libraries [368]. Our tool support configurations to match several leakage model and spatial granularity. As a result, one can use MicroWalk to find intra-cache-line leakages with an arbitrary choice of granularity.

We expect that extending a tool like MicroWalk to find further generic applications' leakages, and not only cryptographic leakage, will be a valuable contribution. Making such tools would be an essential step toward TEE applications' security analysis due to their unique attacker model described in our work. Users use TEEs for several privacy-preserving applications, and it is not clear how attacks like CopyCat would affect the privacy and security of these applications.

7.1.3 Architectural Fixes and Mitigations

Microarchitectural vulnerabilities have resulted in short-term and long-term mitigations at various architectural levels, *i.e.*, hardware, operating systems, and runtime environments. We now discuss some of these efforts.

Blocking precise timing. For most attacks on JavaScript, removing accurate timers from the browser would decrease attackers' chances to exploit Spoiler, MemJam, and Medusa. Indeed, some web browsers have removed timers or distorted them by jitters as a response to attacks [224]. However, such ad-hoc approaches are generally ineffective in the long run; there is a wide range of timers with varying precision available, and removing all of them seems impractical [110, 301].

Similarly, one may try to mitigate the user and network instance of the TPM-Fail attack by disturbing precise time measurement. The OS can add a pre-determined delay to the TPM interface for TPM commands to ensure that it is executed in a constant-time fashion. However, this requires precise estimation of an upper bound for the execution time for these operations. This estimation is not trivial since the execution times vary among different TPMs. Also, concerning the physical adversarial model of secure co-processors, this approach would only make sense if deployed inside the TPM.

Tweaking SMT. Some attacks such as MEDUSA and MEMJAM directly affect simultaneous multithreading (SMT) like hyperthreading. In particular, there is no way to prevent MDS attacks on CPUs before the 10^{th} Intel generation (Ice Lake) when hyperthreading is enabled. Although Intel claims that workloads can run securely with hyperthreading if group scheduling is implemented [157], we are not aware of any commodity operating system implementing group scheduling. Similarly, MEMJAM highlights the combined effect of hyperthreading and false dependency and its impact on application security. Hence, we stress that hyperthreading has to be disabled to prevent MEDUSA and MEMJAM entirely. Disabling SMT for microarchitectures affected by vulnerabilities like above is not generally a desirable outcome due to loss of performance and net gain. As a result, hardening SMT against microarchitectural side channels for future hardware has got attention [337].

Microcode patches for MDS. We have not experienced a vendor like Intel issuing microcode patches or a clear mitigation plan for microarchitectural side channels leaking memory access patterns. On the other hand, to prevent the exploitation of MDS attacks including MEDUSA during context switching, Intel suggests a microcode update that retrofits the `VERW` instruction with the side effect that it clears the store buffer, fill buffer and load ports. Schwarz et al. [300] have shown that ZombieLoad can, unfortunately, circumvent this mitigation. As a result, the only practical solution is to flush the L1 data cache across context switching as well. However, flushing the store buffer, fill buffer, load ports, and L1 data cache on every privilege-level context switching incurs non-negligible performance overhead.

Although more recent CPUs, like the *Cascade Lake* and *Coffee Lake* (9^{th} generation) are promised to be MDS resistant, there are still variants of ZombieLoad which work on these CPUs by leveraging microcode assists caused by Intel TSX. Similarly, MEDUSA can benefit from TSX, and it works on these CPUs. Hence, even on some MDS-resistant CPUs, Intel TSX has to be disabled to ensure that MDS cannot leak any data. While Intel TSX cannot be disabled directly, a workaround is to ensure that all TSX transactions abort immediately by setting the `MSR_TSX_FORCE_ABORT` model-specific register. As a consequence, Intel TSX cannot be used for fault suppression any more.

MDS-testing microcode patches with Transynther. Intel announced that the newest microarchitectures, namely Cascade Lake and Ice Lake, were not affected by MDS. While Cascade Lake turned out to be vulnerable to the ZombieLoad v2 MDS attack (also known as TAA), Ice Lake was not affected by this attack. Using TRANSYNTHER, we show a variant of MDS attack, also known as Fallout [61], that works on Ice Lake CPUs. Ice Lake is reported to be unaffected by all MDS attacks [167, 169]. Intel has also explicitly listed Ice Lake processors as not being vulnerable to LVI-SB [349], which exploits Fallout for Load Value Injection [168]. TRANSYNTHER automatically synthesized this finding. Based on these findings, we analyze different microcodes regarding this issue, showing that only microcode versions after January 2020 prevent exploiting the vulnerability. These results show that TRANSYNTHER is a valuable tool to find new variants and test for regressions possibly introduced with microcode updates.

We ran TRANSYNTHER on a Core i5-1035G1 CPU with the latest microcode shipped with Ubuntu 18.04, version 0x48. After running for about 5000 iterations, TRANSYNTHER reported store-to-load-forwarding leakage due to 4K aliasing of store addresses with a faulty memory load. Fallout initially exploited this behavior to bypass KASLR and leak cryptographic keys from the kernel space [61]. Based on the generated proof-of-concept, we produced a minimal working example to analyze the auto-generated proof of concept that triggers this condition manually. We noticed that store buffer leakage on Ice Lake only works with memory load operations that suffer a permission failure due to accessing privileged memory (cleared US bit) or accessing a memory page with wrong protection keys [62]. Based on the systematization of Canella et al. [62], and to the best of our knowledge, we conclude that Ice Lake is only vulnerable to Meltdown-US or Meltdown-MPK attacks.

One of the observations from TRANSYNTHER is that the leakage rate increases significantly if the target store address is flushed from the cache. We observe the same behavior for other instructions that modify the cache state. Specifically, executing `lock incl` on the store address leads to an even higher leakage rate than flushing the store address using `clflush`. Listing 11 shows our simplified proof of concept that demonstrates store buffer leakage on the Ice Lake microarchitecture. Uncommenting Line 6 or 7 modifies the store address's cache state, resulting in a faster leakage. If we do not modify the cache state, we observe a very slow leakage of approximately 1 B/s. As we can see in Table 7.1, with approximately 750 B/s, the leakage rate is significantly higher when using `lock inc` instruction (Line 19) to modify the cache state.

We also tested the proof-of-concept on various microcode versions on the Ice Lake CPU. As not all issued microcodes are officially available by CPU vendors, we used a crowd-sourced repository of available microcodes [119]. For our analysis, we applied ten different compatible microcode versions, i.e., microcodes that match the CPUID of our target CPU. As we can see in Table 7.1, all Intel microcodes until mid-November are vulnerable to store buffer data sampling, although Ice Lake should fundamentally be resistant against all MDS attacks.

Listing 11 Proof of concept for store buffer data sampling on IceLake.

```
/** asm.S **/
   .global s_faulty_load
s_faulty_load:
 lea address_normal+0x4822, %r14    // Store address
 lea address_supervisor+0x822, %r15 // Load address (4K alias)
//clflush (%r14)                    // Uncomment to modify cache state
//lock;incl (%r14)                  // Uncomment to modify cache state
 movb $0x41, (%r14)                 // Store
 movb (%r15), %al                   // Faulty Load
 lea oracles, %r13                  // Encode
 and $0xff, %rax
 shlq $12, %rax
 movb (%r13,%rax,1), %al
 ret
```

Our report shows that Intel Ice lake client processors with early firmware versions are vulnerable to MDS attacks. In discussions with Intel engineers, we were told that mitigations for store buffer data sampling are present in hardware but disabled in early versions of these processors. OEMs and users must apply these latest microcode updates to enable protection against MDS attacks.

Future hardware. Researchers have also suggested several hardware designs in response to some of the microarchitectural attacks. For example, researchers have proposed a custom memory manager [391], relaxed inclusion caches [198] and solutions based on cache allocation technology (CAT) such as Catalyst [225] and vCat [379] to defend against cache attacks. However, known Hardware solutions to defend against cache attacks generally ignore leakages through false dependency. Relaxed inclusion cache is a secure counterpart to the inclusive LLC, which only aims to defend LLC contention [198]. Solutions such as CacheBar [391], Catalyst [225] and vCat [379], which isolate the LLC between different security domains, are not scalable to thwart the MemJam attack, which exploits leakage in the L1 cache. Sanctum [78] is a secure CPU design that uses page coloring to isolate cache. Further, they flush the L1 and TLB cache during context switch from/to secure enclaves. Ozone [24], as a zero timing leakage CPU, aims to defend against such leakages by allocating a constant computational resource to one execution thread per core ignoring the hyper-threading model.

Some processor vendors may revise the hardware design for the memory false dependency checking and resolution to prevent Spoiler and MemJam, but modifying this

MC Version	MC Date	Vulnerable	Leakage (bytes/s)		
			clflush	*lock inc*	Unmodified
0x32 (stock)	2019-07-05	✓	577.87	754.99	1.58
0x36	2019-07-18	✓	148.24	529.84	0.62
0x46	2019-09-05	✓	130.15	695.80	0.11
0x48	2019-09-12	✓	271.69	620.07	0.59
0x50	2019-10-27	✓	96.54	542.10	0.25
0x56	2019-11-05	✓	145.46	751.40	0.08
0x5a	2019-11-19	✓	532.40	645.32	0.70
0x66	2020-01-09	✗	0	0	0
0x70	2020-02-17	✗	0	0	0
0x82	2020-04-22	✗	0	0	0
0x86	2020-05-05	✗	0	0	0

Table 7.1: List of tested microcodes on a Core i5-1035G1 CPU. For vulnerable microcodes, the leakage rate is much higher if the target store is in a modified state, as it is shown by using cache flush and modification instructions. We ran each experiment for two minutes.

component may cause performance impacts if not done carefully. For instance, partial address comparison was a design choice for performance. Full address comparison may address this vulnerability but will also impact performance. Note that microcode patches are generally difficult to be applied to these components as they shape the core of the microarchitecture. Therefore, having a vulnerability at this stage will impact legacy systems for many years to come.

For MDS and transient executions due to the memory subsystem, future CPUs may adopt more advanced resource sharing and partitioning countermeasures to circumvent such attacks [87, 337]. Besides, we have already seen that some of these resources, e.g., the store buffer on Intel Core microarchitecture [2, 187], are statically partitioned and cannot be used in cross-process side-channel attacks. However, as resource sharing is inevitable for core performance, removing the root cause for Meltdown-type attacks seems to be a more fundamental solution. As we discuss in Section 3.4.2, MDS and Meltdown vulnerabilities that affect some of the processor vendors are due to specific optimizations regarding how the pipeline handles microcode assists and pipeline flushes. Some designs may benefit in terms of performance and circuit space from delaying the handling of hazards to a later stage during instruction retirement. Flushing the pipeline as soon as a hazard occurs requires a more complex logic for identifying invalid instructions and flushing the pipeline. Such a design further has to consider all the performance constraints and memory bottlenecks. We expect to see more research in the future on the effect of different strategies for handling such hazards and their security implications.

Concerning future hardware mitigation for CopyCat, recent work [266] proposes modifications to the Intel SGX architecture to rule out page-fault controlled channels by delegating paging decisions to the enclave. The proposed design modifies the CPU to no longer report the faulting page base address to the untrusted OS and to not update "accessed" and "dirty" page-table attributes when in enclave mode. While these modifications would indeed thwart the deterministic spatial dimension of the CopyCat instantiations described in this paper, we expect that adversaries may adapt by resorting to alternative side-channel oracles to construct instruction-granular page access patterns. A particularly promising avenue in this respect would be to combine CopyCat interrupt counting with the distinct timing differences observed for unprotected page-table entries brought into the CPU cache during enclaved execution [353]. Nevertheless, following a long line of microarchitectural attacks abusing interrupts [153, 218, 245, 351, 352], our study provides strong evidence that interrupts may also amplify deterministic controlled-channel leakage and should be taken into account in the enclaved execution threat model. We advocate for architectural changes in the Intel SGX design and further research to rule out interrupt-driven attack surface [56].

7.1.4 Coordinated Vulnerability Disclosure

Responsible disclosure and working with vendors who have access to the required resources to fix these vulnerabilities are essential to countering our findings. We have reported our findings to vendors during several stages of our work as part of coordinated and responsible disclosure. Here we provide a summary of the progress, disclosure timeline, and their response to these reports.

MemJam. We reported MemJam to the Intel Product Security Incident Response Team (PSIRT). They have acknowledged the receipt and confirmed that they would update the Intel IPP library to mitigate MemJam. Here is the timeline in more detail:

- **08/02/2017:** We informed our findings to the Intel PSIRT.
- **08/04/2017:** Intel PSIRT acknowledged the receipt.
- **11/07/2017:** `Safe2Encrypt_RIJ128` was removed from the SGX SDK.
- **11/17/2017:** Intel PSIRT assigned CVE-2017-5737 and confirmed a work-in-progress patch for the IPP library.
- **05/10/2018:** Intel PSIRT published an update for the IPP library with CVE-2018-3691.

Although Intel has mitigated the impact of MemJam on their cryptographic software by avoiding encryption schemes with memory lookups, we are not aware of any vendor's plan to address the hardware's root cause.

Spoiler. We informed the Intel PSIRT of our findings for Spoiler. iPSIRT thanked us for reporting the issue and for the coordinated disclosure. iPSIRT then released the public advisory and CVE. However, they did not disclose any plan to address the root cause of this vulnerability. Here is the time line for the responsible disclosure:

- **12/01/2018:** We informed our findings to iPSIRT.
- **12/03/2018:** iPSIRT acknowledged the receipt.
- **04/09/2019:** iPSIRT released public advisory (INTEL-SA-00238) and assigned CVE (CVE-2019-0162).

Transynther and Medusa. We disclosed our findings regarding Medusa to Intel PSIRT on June 24, 2019. iPSIRT has acknowledged the receipt on the same day. Intel confirmed that the WC is part of the fill buffer, so they will not issue a separate plan for mitigating this attack technique. However, they asked us to keep the paper under embargo until November 12, 2019, as we exploit TSX Asynchronous Abort (TAA, CVE-2019-11135) in several proof of concepts [175].

We have also reported our finding regarding store buffer data leakage on Ice Lake to Intel PSIRT on March 27, 2020. On May 5, 2020, iPSIRT completed the triage of our proof of concept. They replied that the mitigation for store buffer data sampling was not ported correctly to the Ice Lake microarchitecture. As a result, Ice Lake required a microcode patch, which they developed as part of their late November 2019 microcode version 0x5C. In the May 2020 update of Intel's specification update for the 10th Generation Intel Core Processor Family, a new errata, 057, has been added. This errata mentions that the `MDS_NO` bit in `IA32_ARCH_CAPABILITIES` control registers were incorrectly set [179]. Intel requested an embargo until July 14, 2020, to allow enough time for OEMs and their customers to deploy these patches. Intel credited us by updating the advisory regarding Microarchitectural Data Sampling on July 14, 2020 [165]. Later on, during the summer of 2020, we have shared Transynther with Intel engineers and discussed potential ways to integrate this tool into their future pre-silicon and post-silicon security testing future products.

CopyCat. We shared our attack with the Intel PSIRT, who acknowledged that CopyCat leaks side-channel information, but re-iterated that protecting against side channels requires the enclave developer to follow the constant-time coding best practices as advised by Intel [180]. We reported the weaknesses in WolfSSL in Nov. 2019 and provided guidelines for mitigation for the cryptographic libraries, tracked via CVEs 2019-1996{0,1,3} and CVE-2020-7960. We reported our findings to OpenSSL and Libgcrypt teams in Feb. 2020. OpenSSL replaced `BN_gcd` with a constant-time implementation [34] in version 1.1.1e. Libgcrypt issued a similar fix that will appear in version 1.8.6. Later on in Sep. 2020, we discussed with CopyCat with Intel and the potential impact on future products. They have shown interest in mitigating the root cause of CopyCat, but we are not aware of how they will address this class of attacks within the hardware.

TPM-Fail. We informed the Intel PSIRT of our findings regarding Intel fTPM on February 1, 2019. Intel acknowledged receipt on the same day and responded that an outdated version of Intel IPP had been used in the Intel fTPM on February 12, 2019. Intel assigned CVE-2019-11090 and awarded us separately for three vulnerabilities. We informed STMicroelectronics of our findings regarding the TPM chip flaw on May 15, 2019. They acknowledged receipt on May 17, 2019. Later, After patching the firmware for their TPM, STMicroelectronics provided us an updated TPM product that we have verified to be resistant against TPM-Fail. The embargo date for all these issues was set to November 11, 2019.

7.2 Open Problems

This section outlines some of the open problems in this space. We first discuss the limitation of our work for general-purpose software and potential opportunities in this direction (§7.2.1). Then, we discuss other open problems related to expanding such analysis to nonubiquitous and heterogenous microarchitectures (§7.2.2).

7.2.1 General-purpose Software

As its name suggests, microarchitectural cryptanalysis shows the impact of microarchitectural vulnerabilities on specific applications. However, these attacks can fundamentally leak information about general-purpose applications as well. Apart from the data leakage enabled by transient execution attacks, evaluating the impact of microarchitectural side channels on various privacy- and security-critical applications that are not necessarily cryptographic operations is critical for understanding the attack landscape. Lack of this understanding contributes to confusion for designers who care about microarchitectural side channels. For example, a designer may assume that leaking side-channel information is not relevant to a particular processor because customers are not executing cryptographic operations. We have seen some efforts by the community to apply cache attacks to demonstrate key stroke recoveries [215, 222], privacy leakage of web application [136], reconstructing private databases [308], and recovering machine learning models [381]. However, we believe there is still a lot more work to do in this direction:

Automated analysis of general-purpose applications. Despite these manual efforts to demonstrate leakage on applications [134, 215, 308, 381], we have not seen practical tools like MicroWalk to show leakage of any programs in the presence of a particular attack model. A significant challenge in designing such tools is the infinite input space for general-purpose applications. Cryptographic implementations, deep-neural network models, or keystrokes are still applications with a limited input space. Understanding microarchitectural side-channel leakage for data-intensive applications processing arbitrary data and protocol formats is at least as challenging as the automated discovery of traditional software vulnerabilities. The latter is a problem that researchers have been working on for a couple of decades, proposing various techniques based on fuzzing [11, 121], taint analysis [258], or symbolic execution [324]. We are unsure if we can adopt similar techniques to ease the impact evaluation of microarchitectural leakage on general-purpose applications.

Incoporating new attack models. CopyCat is a prime example of an attack that opens up a different threat model and impacts general-purpose applications. Although we have shown a simple example of how this technique can bypass previous code hardening schemes, we did not evaluate its impact on various general-purpose applications. The limitation is that it is challenging to perform the manual analysis we did for cryptographic implementations with domain knowledge for arbitrary applications. However, one may ask Why we should care about the impact of this particular on general-purpose applications. The answer to this question has two components.

First, the powerful system-level threat model applied to CopyCat applies to TEEs other than SGX as well. For example, other researchers have shown high-resolution cache attacks enabled by adversarial OS to ARM TrustZone [287]. TEEs, despite all the side-channel issues, have become a defacto standard for privacy-preserving computing [217], which means developers deploy general-purpose logic inside TEEs. Considering both the use case and the much higher capabilities of attacks on TEEs, we expect to see microarchitectural attacks being efficient on the general-purpose application running inside TEEs.

Second, the specific leakage pattern of an attack like CopyCat has not been concerned in previous attacks on general-purpose computation. Previous demonstrations only focus on the cache access pattern, but the leakage pattern is fundamentally crucial to understand the impact of leakage on these applications. Similarly, there are other kinds of microarchitectural side channels that have different leakage model [104, 246, 380]. Therefore, analysis tools or attack demonstrations should go beyond cache attacks to give a more broad picture of these attacks' impact on general-purpose applications.

7.2.2 Nonubiquitous and Heterogenous Architecture

Automated testing of nonubiquitous processors. In TRANSYNTHER, we mainly focussed on Intel CPUs. While MEDUSA is a vulnerability we only discovered on Intel CPUs, the general approach of TRANSYNTHER applies to different CPUs. We also used TRANSYNTHER on AMD, showing that AMD also forwards data after certain exceptions, a requirement for Meltdown-type attacks. We could not find any variant on AMD that leaks data across a security boundary. We also focused on the similar ubiquitous microarchitectures during our work on MEMJAM and SPOILER. Classifying these vulnerabilities on a standard microarchitecture is beneficial for advancing microarchitectural security. On the other hand, we need similar analysis tools and techniques for nonubiquitous microarchitectures to fulfill security engineering needs.

However, such analysis requires automated tools that scale to several different microarchitectures. For example, we may extend TRANSYNTHER to entirely different microarchitectures, such as ARM or RISC-V. Although the approach is the same, porting TRANSYNTHER to a different instruction set requires a new backend that generates assembly code for the targeted architecture. As our tool is open source, we encourage researchers to port TRANSYNTHER to different architectures to analyze whether they suffer from similar vulnerabilities.

In the same direction, we need tools to automatically analyze other processors for attacks such as Spectre, MEMJAM, or SPOILER. We expect even to see new microarchitectural vulnerabilities that have not been seen before on Intel processors. Such analysis is especially becoming more relevant to the current trend in processor design and the speeding evolution of computer microarchitectures.

Heterogeneity. Heterogenous microarchitectures, combining several different processing technologies such as FPGA, GPU, and CPU into a single system on a chip (SoC), are trending. Depending on the performance requirement and applications, sometimes these designs introduce tightly-connected interfaces, including shared memory resources between different components. In collaboration with Intel, we have started looking into the security of integrated FPGA-CPU systems. Our easier work on JackHammer [367] shows that microarchitectural security has new and exciting challenges for such integrations.

Microarchitectural security for heterogenous microarchitectures are still in infancy, and yet we do not even have a solid understanding of how isolation boundaries should look like for these systems. Adding this new complexity with nonubiquitous systems suggests that the work conducted in these dissertations may expand much further in the next few years in these new avenues.

7.3 Finale

This section concludes this dissertation by providing an assessment check-list for microarchitectural security testing and highlighting the key takeaways.

7.3.1 Assessment check-list

As we have extensively discussed, there are several challenges in the future regarding identifying these vulnerabilities in other systems and efficiently mitigating them. Our findings can not cover and fix all the microarchitectural vulnerabilities in the future or other processors and computing hardware. Even with all the efforts into mitigations and countermeasures described in Section 7.1, there will still be vulnerabilities. Therefore, based on microarchitectural cryptanalysis, we conclude our findings by providing a vulnerability assessment check-list.

Step 1. Identifying shared resources: Security engineers should identify shared components across security boundaries. If the security boundaries are not clear, this is even more difficult, e.g., some heterogeneous systems do not have clear security boundaries. These boundaries also define the trust model. For example, a TEE may share some or all CPU resources with untrusted applications or components. After identifying which components are shared across security boundaries, we should see if they are directly accessible to the software. Note that access to software could include specific instructions, IO operations, or configuration spaces and registers. For example, we have seen that some configuration registers accessible to a privileged adversary enable new attacks on TEEs [251].

Step 2. Prototyping attacks: Prototyping naive covert channels are generally the first step to see if two security boundaries can create a noncanonical communication channel. After that, we can create a proof-of-concept attacking an already well-understood and simple victim application or workload that gives more insights into whether one can turn the identified shared component into an attack vector. At this stage, combining several different known attack techniques may reveal new violations of security boundaries. It is crucial to remember that attackers can always combine multiple architectural and microarchitectural vulnerabilities [224, 349]. This step may reject the hypothesis that there will be any security issue associated with a shared component. Alternatively, we may see some potential sources of leakage, but we can not always be sure about its security implications. Therefore, it is crucial to combine this step forth and back with the next step.

Step 3. Identifying security-critical software: With a proof-of-concept in hand, we can identify the realistic and security-critical software running on this processor. In our work, we have mostly identified cryptographic implementations running on the CPU. However, for general-purpose superscalar CPUs, it is intuitive to assume that any data-processing application may suffer from these vulnerabilities. Additionally, a complex SoC may include several customized accelerators for different purposes. It is important to remember that even though cryptographic implementations are generally the most vulnerable, they will not be the only affected software.

Step 4. Demonstrating the impact: Without Demonstrating the impact with realistic case studies, it is generally not clear what should be mitigated or if one should prioritize addressing potential weaknesses. A working proof of concept sheds light on understanding the problems accurately and responds appropriately. We discussed several of the failed countermeasures in our work due to such a lack of end-to-end understanding of a vulnerability from a high-level hypothesis to low-level engineering challenges. Demonstrating impact answers critical questions such as:

1. What can be compromised?
2. How efficient is the attack?
3. What are the requirements for the attackers?

These questions essentially provide us enough information for the next stage. One reason vendors and industry did not take microarchitectural vulnerabilities seriously in the earlier days was this lack of impact demonstration. However, we see that thanks to recent findings of transient execution attacks and several iterations of MDS, the industry essentially have a more clear picture if they should allocate resources to these problems. Our microarchitectural cryptanalysis essentially takes this understanding to the next level by demonstrating end-to-end proof of concepts. However, similar techniques may apply to general-purpose applications [308].

Step 5. Prioritizing mitigation plans: We can work across several engineering and research teams developing mitigating plans based on the attack's impact. Short-term mitigation generally constitutes software-based hacks or more complex compiler-level designs. Ultimately, depending on the severity of these issues and performance impact, future hardware designs may incorporate the findings into fixing these issues altogether or provide some hardware support for the software to deal with these vulnerabilities more efficiently.

Putting it together. We may not always take the steps described above in the same order, but it is crucial to go over several iterations based on our experience. Almost every shared computing infrastructure has at least tens of different shared components and interfaces, so it would be naive to assume that a processor is inherently secure against software-based microarchitectural attacks at first glance. Significantly, the first few steps are much trivial when engineers have access to design artifacts. In our work, we had to spend a lot of time on the first two steps due to the closed-source nature of commodity CPUs we have studied in this dissertation. We have developed several tools for precise microbenchmarks, automated microarchitectural and software analysis tools, and simulation during each of these stages. Based on this experience, we expect practitioners always to remember the importance of automated tooling to speed up such analysis.

7.3.2 Conclusion

This dissertation has expanded the understanding of microarchitectural attacks by introducing several new attack vectors. Furthermore, we have addressed several uncertainties about the security implications of commodity microarchitectures concerning several threat models, including trust and isolation in the presence of system adversaries. TRANSYNTHER highlights the importance of automated vulnerability testing and analysis for hardware and microarchitectural vulnerabilities. Although TRANSYNTHER is an academic prototype, it still proves to be a valuable tool for automated hardware testing. The newly reported MDS vulnerability would not have gone unnoticed on Ice Lake if the hardware's earlier prototypes were tested using such tools. Furthermore, OEMs could have tested these vulnerabilities before shipping consumer laptop with a vulnerable microcode update. We have shown these vulnerabilities by devising new algorithmics attacks combined with these leakages, introducing our findings under *microarchitectural cryptanalysis*.

Our work shows that software-based side-channel attacks are a practical threat to complex computing systems, while our reports help future CPUs and cryptographic software become more secure. As our findings suggest, these vulnerabilities affect several different threat models, including network adversaries, local adversaries with the least privilege, and system adversaries attacking hardware-based trusted computing technologies. Consequently, it is crucial for designers to understanding and treats these threat models properly. An important key takes away from vulnerabilities like CopyCat, and TPM-Fail is that sometimes these vulnerabilities occur because of porting a previous design to a different threat model. In particular, Intel SGX relies on an ISA and a legacy architecture not designed for the TEE model. Similarly, several of the cryptographic Implementations used today in such trust models have been designed with a different threat model and a much-limited understanding of potential attack vectors. However, there is no guarantee that new designs with all these considerations would be immune to microarchitectural attacks. That is why automated tools such Transynther and MicroWalk plays an essential role in understanding the root cause and impact of these issues better, verifying hardware mitigations, and automated testing of both the architecture and software. Furthermore, these tools help us identify vulnerabilities at a larger scale, as our finding suggests that manual effort to find these attack vectors does not scale to nonubiquitous systems.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[2] Jeffery M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, and Paul D Madland. Method and Apparatus for Performing a Store Operation, April 2002. US Patent 6,378,062.

[3] Jeffrey M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, Paul D Madland, David B Papworth, and Michael A Fetterman. Method and Apparatus for Dispatching and Executing a Load Operation to Memory, February 1998. US Patent 5,717,882.

[4] Can Acar, Arvind Krishnaswamy, and Robert Turner. Code pointer authentication for hardware flow control, December 2016. US Patent 9,514,305.

[5] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New Results on Instruction Cache Attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2010.

[6] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. *Cryptography and Coding*, 2007.

[7] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2007.

[8] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys via Branch Prediction. In *Cryptographers' Track at the RSA Conference*, 2007.

[9] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[10] Advanced Micro Devices. Software Optimization Guide for AMD Family 17h Processors. https://developer.amd.com/resources/developer-guides-manuals/, 2017.

[11] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Accessed: December 10, 2020.

[12] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The EM side—channel (s). In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2002.

[13] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. Breakthrough AES Performance with Intel AES New Instructions. *White paper, June*, 2010.

[14] Alejandro Cabrera Aldaya and Billy Bob Brumley. When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020.

[15] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[16] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-Timing Attacks on RSA Key Generation. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019.

[17] Alejandro Cabrera Aldaya, Alejandro J Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended Euclidean algorithm. *Journal of Cryptographic Engineering*, 2017.

[18] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying Side Channels Through Performance Degradation. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[19] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time Implementations. In *USENIX Security Symposium*, 2016.

[20] AMD. Speculation Behavior in AMD Micro-Architectures. https://www.amd.com/system/files/documents/security-whitepaper.pdf, 2019.

[21] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On Subnormal Floating Point and Abnormal Timing. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[22] ARM. Security technology building a secure system using trustzone technology (white paper). *ARM Limited*, 2009.

[23] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS with SSLv2. In *USENIX Security Symposium*, 2016.

[24] Zelalem Birhanu Aweke and Todd Austin. Øzone: Efficient Execution with Zero Timing Leakage for Modern Microarchitectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[25] 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. *ACM SIGPLAN Notices*, 2016.

[26] Sundeep Bajikar. Trusted Platform Module (TPM) based Security on Notebook PCs - White Paper. *Mobile Platforms Group Intel Corporation*, 2002.

[27] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography*, 2006.

[28] Erick Bauman and Zhiqiang Lin. A Case for Protecting Computer Games With SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, 2016.

[29] BearSSL. BearSSL Constant-Time Crypto. https://www.bearssl.org/constanttime.html. Accessed: December 10, 2020.

[30] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A Taint Based Approach for Smart Fuzzing. In *IEEE International Conference on Software Testing, Verification and Validation*, 2012.

[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 *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2014.

[32] Daniel J Bernstein. Cache-timing attacks on AES. http://palms.ee.princeton.edu/system/files/Cache-timing+attacks+on+AES.pdf, 2005.

[33] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[34] Daniel J Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019.

[35] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[36] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[37] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential Fault Attacks on Elliptic Curve Cryptosystems. In *Annual International Cryptology Conference*, 2000.

[38] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying Constant-Time Implementations by Abstract Interpretation. *Journal of Computer Security*, 2019.

[39] Daniel Bleichenbacher. Experiments with DSA. *CRYPTO 2005–Rump Session*, 2005.

[40] Johannes Blömer and Alexander May. New Partial Key Exposure Attacks on RSA. In *Annual International Cryptology Conference*, 2003.

[41] David G Boak. A History of US Communications Security (Volumes I and II). *National Security Agency*, 1973.

[42] Dan Boneh and Glenn Durfee. Cryptanalysis of RSA with private key d less than N/sup 0.292. *IEEE transactions on Information Theory*, 2000.

[43] Dan Boneh, Glenn Durfee, and Yair Frankel. An Attack on RSA Given a Small Fraction of the Private Key Bits. In *International Conference on the Theory and Application of Cryptology and Information Security*, 1998.

[44] Dan Boneh, Shai Halevi, and Nick Howgrave-Graham. The Modular Inversion Hidden Number Problem. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2001.

[45] Dan Boneh and Ramarathnam Venkatesan. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In *Advances in Cryptology*, 1996.

[46] Marcus Brandenburger and Christian Cachin. Challenges for Combining Smart Contracts with Trusted Computing. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018.

[47] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[48] 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.

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

[50] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software Mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006.

[51] Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. Mitigating cache/timing based side-channels in AES and RSA software implementations. In *RSA Conference 2006 session DEV-203*, 2006.

[52] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. CacheShield: Detecting Cache Attacks Through Self-Observation. In *ACM Conference on Data and Application Security and Privacy*, 2018.

[53] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks are Still Practical. In *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2011.

[54] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In *USENIX Security Symposium*, 2003.

[55] David Burns. Pre-Silicon Validation of Hyper-Threading Technology. *Intel Technology Journal*, 2002.

[56] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Provably Secure Isolation for interruptible Enclaved Execution on Small Microprocessors. In *IEEE Computer Security Foundations Symposium (CSF)*, 2020.

[57] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. BIOS Chronomancy: Fixing the Core Root of Trust for Measurement. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[58] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J Cabrera Sarmiento, and Santiago Sánchez-Solano. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *International Journal of Circuit Theory and Applications*, 2017.

[59] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security (TISSEC)*, 2008.

[60] Carole Cadwalladr and Emma Graham-Harrison. Revealed: 50 million Facebook profiles harvested for Cambridge Analytica in major data breach. *The guardian*, 2018.

[61] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, and et al. Fallout: Leaking Data on Meltdown-Resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[62] 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*, 2019.

[63] Adrian L Carbine, Gary L Brown, and Donald D Parker. Decoder for decoding multiple instructions in parallel, May 1997. US Patent 5,630,083.

[64] George J Carrette. CRASHME: Random input testing, 1996.

[65] Chandler Carruth. Rfc: Speculative load hardening (a spectre variant1 mitigation). `https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html`, 2018.

[66] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-Time Foundations for the New Spectre Era. In *Programming Language Design and Implementation (PLDI)*, 2020.

[67] David Challener. Trusted Platform Module. In *Encyclopedia of Cryptography and Security*, 2011.

[68] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2019.

[69] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[70] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, 2011.

[71] Lily Chen, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on Post-quantum Cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016.

[72] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[73] Brian Chess and Gary McGraw. Static Analysis for Security. In *IEEE Symposium on Security and Privacy (S&P)*, 2004.

[74] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 2016.

[75] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[76] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 1997.

[77] Jean-Sébastien Coron, Marc Joye, David Naccache, and Pascal Paillier. Universal Padding Schemes for RSA. In *Annual International Cryptology Conference*, 2002.

[78] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, 2016.

[79] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, 2000.

[80] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[81] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Science & Business Media, 2013.

[82] 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 *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018.

[83] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version. *Journal of Cryptographic Engineering*, 2014.

[84] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[85] Bert den Boer, Kerstin Lemke, and Guntram Wicke. A DPA attack against the modular reduction within a CRT implementation of RSA. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2002.

[86] Dorothy E Denning. An Intrusion-Detection Model. *IEEE Transactions on software engineering*, 1987.

[87] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security Symposium*, 2020.

[88] Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) Protocol version 1.2. 2008.

[89] Whitfield Diffie and George Ledin. SMS4 Encryption Algorithm for Wireless Networks. *IACR Cryptology ePrint Archive*, 2008.

[90] Pei Dingyi, Salomaa Arto, and Ding Cunsheng. *Chinese Remainder Theorem: Applications In Computing, Coding, Cryptography*. World Scientific, 1996.

[91] 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.

[92] Jack Doweck. Inside Intel® Core Microarchitecture. In *IEEE Hot Chips 18 Symposium (HCS)*, 2006.

[93] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. In *Programming Language Design and Implementation (PLDI)*, 2017.

[94] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The Matter of Heartbleed. In *Proceedings of conference on internet measurement conference*, 2014.

[95] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[96] Donald Eastlake and Paul Jones. US secure hash algorithm 1 (SHA1), 2001.

[97] EMVCo. EMVCo overview. https://www.emvco.com/about/overview/. Accessed: December 10, 2020.

[98] EMVCo. Integrated Circuit Card Specifications for Payment Systems – Book 2: Security and Key Management, Version 4.3, 2011.

[99] Mark Ermolov and Maxim Goryachy. How to hack a turned-off computer, or running unsigned code in intel management engine. *Black Hat Europe*, 2017.

[100] Mark Ermolov and Maxim Goryachy. Where There's a JTAG, There's a way: Obtaining full system access via USB. *White Paper*, 2017. Accessed: December 10, 2020.

[101] Mark Ermolov and Maxim Goryachy. Intel VISA: Through the Rabbit Hole. *Black Hat Asia*, 2019.

[102] Matthias Ernst, Ellen Jochemsz, Alexander May, and Benne De Weger. Partial Key Exposure Attacks on RSA Up to Full Size Exponents. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2005.

[103] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *IEEE/ACM International Symposium on Microarchitecture*, 2016.

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

[105] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[106] N. J. Al Fardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[107] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, 2012.

[108] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building Diverse Computer Systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, 1997.

[109] Fortanix. Self-Defending Key Management Service with Intel® Software Guard Extensions. https://www.fortanix.com/assets/SGXwhitepaper/Fortanix_SDKMS_with_Intel_SGX_Whitepaper.pdf. Accessed: December 10, 2020.

[110] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[111] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[112] Patrick Gallagher. Digital Signature Standard (DSS). *Federal Information Processing Standards Publications, volume FIPS*, 2013.

[113] Cesar Pereida García and Billy Bob Brumley. Constant-Time Callees with Variable-Time Callers. In *USENIX Security Symposium*, 2017.

[114] Amaury Gauthier, Clément Mazin, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Enhancing fuzzing technique for OKL4 syscalls testing. In *International Conference on Availability, Reliability and Security (ARES)*, 2011.

[115] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 2018.

[116] Qian Ge, Yuval Yarom, Frank Li, and Gernot Heiser. Contemporary Processors Are Leaky–and There's Nothing You Can Do About It. *The Computing Research Repository. arXiv*, 2016.

[117] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by Key-Extraction Cache Attacks from Portable Code. In *International Conference on Applied Cryptography and Network Security*, 2018.

[118] Craig Gentry and Shai Halevi. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *Annual international conference on the theory and applications of cryptographic techniques*, 2011.

[119] Github. CPUMicrocodes: Intel, AMD, VIA & Freescale CPU Microcode Repositories. https://github.com/platomav/CPUMicrocodes, May 2020.

[120] Cezary Glowacz, Vincent Grosso, Romain Poussier, Joachim Schueth, and François-Xavier Standaert. Simpler and More Efficient Rank Estimation for Side-Channel Security Assessment. In *International Workshop on Fast Software Encryption*, 2015.

[121] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated Whitebox Fuzz Testing. In *Network and Distributed Systems Security (NDSS) Symposium*, 2008.

[122] Google. Shielded VM. `https://cloud.google.com/security/shielded-cloud/shielded-vm#vtpm`, 2019. Accessed: December 10, 2020.

[123] 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.

[124] Seena Gressin. The equifax data breach: What to do. *Federal Trade Commission*, 2017.

[125] 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 Symposium on Security and Privacy (S&P)*, 2018.

[126] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[127] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[128] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

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

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

[131] S. Gueron and V. Krasnov. SM4 acceleration processors, methods, systems, and instructions, December 2016. US Patent 9,513,913.

[132] Shay Gueron. Memory Encryption for General-Purpose Processors. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[133] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[134] Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. Cache-Based Application Detection in the Cloud Using Machine Learning. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[135] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. *arXiv preprint arXiv:1907.03651*, 2019.

[136] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *Computer Security – ESORICS 2017*, 2017.

[137] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the Linux random number generator. In *IEEE Symposium on Security and Privacy (S&P)*, 2006.

[138] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution. In *International Symposium on Engineering Secure Software and Systems*, 2018.

[139] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. In *Programming Language Design and Implementation (PLDI)*, 2017.

[140] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *USENIX Annual Technical Conference (ATC)*, 2017.

[141] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, 2006.

[142] Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim. A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping. In *27th USENIX Security Symposium (USENIX Security 18)*, August 2018.

[143] Darrel Hankerson, Julio López Hernandez, and Alfred Menezes. Software implementation of elliptic curve cryptography over binary fields. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2000.

[144] Dan Harkins, Dave Carrel, et al. The Internet Key Exchange (IKE). Technical report, RFC 2409, november, 1998.

[145] Nadia Heninger and Hovav Shacham. Reconstructing RSA Private Keys from Random Key Bits. In *Annual International Cryptology Conference*, 2009.

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

[147] Martin Hlaváč and Tomáš Rosa. Extended Hidden Number Problem and Its Cryptanalytic Applications. In *Selected Areas in Cryptography*, 2007.

[148] Jann Horn. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[149] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018.

[150] George Hotz. Console hacking 2010-ps3 epic fail. In *27th Chaos Communications Congress*, 2010.

[151] Nick A Howgrave-Graham and Nigel P. Smart. Lattice Attacks on Digital Signature Schemes. *Designs, Codes and Cryptography*, 23(3), 2001.

[152] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[153] 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 *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020.

[154] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive*, 2015.

[155] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2016.

[156] Intel. Affected Processors: Transient Execution Attacks & Related Security Issues by CPU. https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model. Accessed: December 10, 2020.

[157] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling. https://www.epanorama.net/blog/2019/05/17/deep-dive-intel-analysis-of-microarchitectural-data-sampling/. Accessed: June 21, 2020.

[158] Intel. Intel IPP Crypto Library (commit ad2ad95). https://github.com/intel/ipp-crypto.

[159] Intel. Intel IPP linkage models - quick reference guide. https://software.intel.com/content/www/us/en/develop/articles/intel-integrated-performance-primitives-intel-ipp-intel-ipp-linkage-models-quick-reference-guide.html. Accessed: December 10, 2020.

[160] Intel. Intel Lehmer'c GCD Implementation sources/ippcp/pcpbnarithgcd.c. https://github.com/intel/ipp-crypto/blob/b6848dc/sources/ippcp/pcpbnarithgcd.c#L54. Accessed: December 10, 2020.

[161] Intel. Intel SGX SSL. https://github.com/intel/intel-sgx-ssl. Accessed: December 10, 2020.

[162] Intel. Intel(R) Software Guard Extensions for Linux* OS. https://github.com/01org/linux-sgx. Accessed: December 10, 2020.

[163] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf. Accessed: December 10, 2020.

[164] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/en-us/articles/intel-sdm. Accessed: December 10, 2020.

[165] Intel. Microarchitectural Data Sampling Advisory (INTEL-SA-00233). https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html. Accessed: July 14, 2020.

[166] Intel. Pin, Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. Accessed: December 10, 2020.

[167] Intel. Processors Affected by Microarchitectural Data Sampling. https://web.archive.org/web/20200621020512/https://software.intel.com/security-software-guidance/api-app/insights/processors-affected-microarchitectural-data-sampling. Accessed: June 21, 2020.

[168] Intel. Processors Affected: Load Value Injection. https://web.archive.org/web/20200519114503/https://software.intel.com/security-software-guidance/insights/processors-affected-load-value-injection. Accessed: May 19, 2020.

[169] Intel. Side Channel Mitigation by Product CPU Model. https://web.archive.org/web/20200523134452/https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html. Accessed: May 23, 2020.

[170] Intel. Understanding CPU Dispatching in the Intel® IPP Libraries. https://software.intel.com/content/www/us/en/develop/articles/understanding-cpu-optimized-code-used-in-intel-ipp.html. Accessed: December 10, 2020.

[171] Intel. Write Combining Memory Implementation Guidelines. https://download.intel.com/design/PentiumII/applnots/24442201.pdf, 1998.

[172] Intel. Intel 64 Architecture Memory Ordering White Paper. http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf, 2008. Accessed: December 10, 2020.

[173] Intel. Copying Accelerated Video Decode Frame Buffers. https://software.intel.com/content/www/us/en/develop/articles/copying-accelerated-video-decode-frame-buffers.html, 2015.

[174] Intel. Intel Software Guard Extensions Developer Guide. https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top.html, 2017.

[175] Intel. Deep Dive: Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort, 2019.

[176] Intel. Developer Reference for Intel Integrated Performance Primitives Cryptography. https://software.intel.com/en-us/ipp-crypto-reference, 2019. Accessed: December 10, 2020.

[177] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[178] Intel. Intel Quark Microcontrollers. https://www.intel.com/content/www/us/en/embedded/products/quark/overview.html, 2019. Accessed: December 10, 2020.

[179] Intel. 10th Generation Intel Core Processor Families Specification Update. https://intel.ly/31x6BcJ, May 2020.

[180] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. https://software.intel.com/security-software-guidance/secure-coding/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations, 2020.

[181] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun Kanuparthi, Thomas Eisenbarth, and Berk Sunar. Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries. *arXiv preprint arXiv:1709.01552*, 2017.

[182] 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 Symposium on Security and Privacy (S&P)*, 2015.

[183] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *Euromicro Conference on Digital System Design (DSD)*, 2015.

[184] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[185] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping Microarchitectural Attacks Before Execution. *IACR Cryptology ePrint Archive*, 2016.

[186] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[187] 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.

[188] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[189] Gabriel Campana Jean-Baptiste Bedrune. Everybody be Cool, This is a Robbery! *Black Hat USA*, 2019.

[190] Moritz Jodeit and Martin Johns. USB Device Drivers: A Stepping Stone into Your Kernel. In *IEEE European Conference on Computer Network Defense*, 2010.

[191] Don Johnson, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 2001.

[192] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. *White Paper*, 2016.

[193] Dave Jones. Trinity: A system call fuzzer. In *Ottawa Linux Symposium*, 2011.

[194] Ramesh Karri, Jeyavijayan Rajendran, Kurt Rosenfeld, and Mohammad Tehranipoor. Trustworthy Hardware: Identifying and Classifying Hardware Trojans. *Computer*, 2010.

[195] Timo Kasper, David Oswald, and Christof Paar. EM Side-Channel Attacks on Commercial Contactless Smartcards Using Low-Cost Equipment. In *International Workshop on Information Security Applications*, 2009.

[196] Bernhard Kauer. OSLO: Improving the Security of Trusted Computing. In *USENIX Security Symposium*, 2007.

[197] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *Cryptology and Network Security*, 2016.

[198] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In *Design Automation Conference (DAC)*, 2017.

[199] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium*, 2014.

[200] Deokjin Kim, Daehee Jang, Minjoon Park, Yunjong Jeong, Jonghwan Kim, Seokjin Choi, and Brent Byunghoon Kang. SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure. *computers & security*, 2019.

[201] Jeremie S Kim, Minesh Patel, A Giray Yaglikci, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. *arXiv preprint arXiv:2005.13121*, 2020.

[202] 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 *ACM SIGARCH Computer Architecture News*, 2014.

[203] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[204] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[205] Donald E Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 2014.

[206] Cetin K Koç. Analysis of Sliding Window Techniques for Exponentiation. *Computers & Mathematics with Applications*, 1995.

[207] 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 Symposium on Security and Privacy (S&P)*, 2019.

[208] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to Differential Power Analysis. *Journal of Cryptographic Engineering*, 2011.

[209] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology*, 1996.

[210] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 1997.

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

[212] Steffen Kosinski, Fernando Latorre, Niranjan Cooray, Stanislav Shwartsman, Ethan Kalifon, Varun Mohandru, Pedro Lopez, Tom Aviram-Rosenfeld, Jaroslav Topp, Li-Gao Zei, et al. Store forwarding for data caches, November 2016. US Patent 9,507,725.

[213] Evgeni Krimer, Guillermo Savransky, Idan Mondjak, and Jacob Doweck. Counter-based memory disambiguation techniques for selectively predicting load/store conflicts, October 2013. US Patent 8,549,263.

[214] Klaus Kursawe, Dries Schellekens, and Bart Preneel. Analyzing trusted platform communication. In *In: ECRYPT Workshop, CRASH – CRyptographic Advances in Secure Hardware*, 2005.

[215] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[216] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[217] Christoph Lambert, Maria Fernandes, Jérémie Decouchant, and Paulo Esteves-Verissimo. MaskAl: Privacy preserving masked reads alignment using intel SGX. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2018.

[218] 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.

[219] Oded Lempel. System for speculative branch target prediction having a dynamic prediction history buffer and a static prediction history buffer, November 1999. US Patent 5,978,909.

[220] A. K. Lenstra, H. W. Lenstra, and L. Lovasz. Factoring Polynomials with Rational Coefficients. *MATH. ANN*, 261:515–534, 1982.

[221] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. Pac it up: Towards pointer integrity using arm pointer authentication. In *USENIX Security Symposium*, 2019.

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

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

[224] 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.

[225] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[226] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[227] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[228] Stefan Mangard. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In *International Conference on Information Security and Cryptology*, 2002.

[229] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, volume 31. Springer Science & Business Media, 2008.

[230] Debbie Marr, Frank Binns, D Hill, Glenn Hinton, D Koufaty, et al. Hyper-threading technology in the netburst® microarchitecture. *Hot Chips*, 2002.

[231] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing system virtual machines. In *19th International Symposium on Software Testing and Analysis*, 2010.

[232] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *Symposium on Software Testing and Analysis*, 2009.

[233] 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 *Network and Distributed Systems Security (NDSS) Symposium*, 2017.

[234] Alexander May. New RSA Vulnerabilities Using Lattice Reduction Methods. *PhD Dissertation*, 2003.

[235] Manuel Mendonça and Nuno Neves. Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities. In *IEEE European Dependable Dependable Computing Conference (EDCC)*, 2008.

[236] Thomas S. Messerges, Ezzat A. Dabbish, and Robert H. Sloan. Examining Smart-Card Security Under the Threat of Power Analysis Attacks. *IEEE Trans. Comput.*, May 2002.

[237] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *USENIX Security Symposium*, 2014.

[238] Microsoft. How Windows 10 uses the Trusted Platform Module. https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/how-windows-uses-the-tpm, 2019. Accessed: December 10, 2020.

[239] Microsoft. Support for generation 2 VMs (preview) on Azure. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/generation-2, 2019. Accessed: December 10, 2020.

[240] Microsoft. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2020. Accessed: December 10, 2020.

[241] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading Kernel Writes From User Space. *arXiv:1905.12701*, 2019.

[242] Chris Mitchell. *Trusted computing*, volume 6. Iet, 2005.

[243] Ahmad Moghimi. Side-Channel Attacks on Intel SGX: How SGX Amplifies The Power of Cache Attack. *MS Thesis*, 2017.

[244] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *Cryptographers' Track at the RSA Conference*, 2018.

[245] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[246] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations. *International Journal of Parallel Programming*, 2019.

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

[248] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *USENIX Security Symposium*, 2020.

[249] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *USENIX Security Symposium*, 2020.

[250] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE Bites: Exploiting the SSL 3.0 Fallback. *Security Advisory*, 2014.

[251] 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 Symposium on Security and Privacy (S&P)*, 2020.

[252] San Murugesan. Understanding Web 2.0. *IT professional*, 2007.

[253] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Muneeb Yousaf, Umer Farooq, Vianney Lapotre, and Guy Gogniat. Machine Learning For Security: The Case of Side-Channel Attack Detection at Run-time. In *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2018.

[254] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *USENIX Security Symposium*, 2020.

[255] National Institute of Standards and Technology. Federal Information Processing Standards (FIPS) Publication 46-3 – Data Encryption Standard (DES). https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf, 1999.

[256] National Institute of Standards and Technology. Update to Current Use and Deprecation of TDEA. https://csrc.nist.gov/News/2017/Update-to-Current-Use-and-Deprecation-of-TDEA, 2017.

[257] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[258] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Network and Distributed Systems Security (NDSS) Symposium*, 2005.

[259] Nguyen and Shparlinski. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *Journal of Cryptology*, 2002.

[260] Phong Q Nguyen and Igor E Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Designs, codes and cryptography*, 2003.

[261] Phong Q. Nguyen and Damien Stehlé. LLL on the Average. In *Algorithmic Number Theory*, 2006.

[262] Phuong Ha Nguyen, Chester Rebeiro, Debdeep Mukhopadhyay, and Huaxiong Wang. Improved differential cache attacks on SMS4. In *International Conference on Information Security and Cryptology*, 2012.

[263] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type Vulnerabilities to the Surface. In *USENIX Security Symposium*, 2019.

[264] openssl.com. OpenSSL Cryptography and SSL/TLS Toolkit). https://www.openssl.org/, 2020.

[265] 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 SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[266] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing Controlled Channels with Self-Paging Enclaves. In *EuroSys*, 2020.

[267] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers' Track at the RSA Conference*, 2006.

[268] D Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 2003.

[269] Colin Percival. Cache missing for fun and profit, 2005.

[270] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make Sure DSA Signing Exponentiations Really Are Constant-Time. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[271] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security Symposium*, 2006.

[272] Diego Perez-Botero, Jakub Szefer, and Ruby B Lee. Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers. In *Proceedings of international workshop on Security in cloud computing*, 2013.

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

[274] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *USENIX Security Symposium*, 2020.

[275] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. In *Smart Card Programming and Security*, 2001.

[276] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *USENIX Security Symposium*, 2016.

[277] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium*, 2015.

[278] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, Precise, and Fast Floating-Point Operations on x86 Processors. In *USENIX Security Symposium*, 2016.

[279] Eric Rescorla and Tim Dierks. The transport layer security (TLS) protocol version 1.3. RFC 8446, DOI 10.17487/RFC8446, August 2018, 2018.

[280] Eric Rescorla et al. Diffie-Hellman Key Agreement Method. Technical report, RFC 2631, June, 1999.

[281] 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 SIGSAC Conference on Computer and Communications Security (CCS)*, 2009.

[282] Matthieu Rivain. Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves. *IACR Cryptology ePrint Archive*, 2011.

[283] Ronald Rivest and S Dusse. The MD5 message-digest algorithm, 1992.

[284] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 1978.

[285] Tanja Römer and Jean-Pierre Seifert. Information leakage attacks against smart card implementations of the elliptic curve digital signature algorithm. In *International Conference on Research in Smart Cards*, 2001.

[286] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *IEEE Symposium on Security and Privacy*, 2019.

[287] Keegan Ryan. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[288] Keegan Ryan. Return of the Hidden Number Problem. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019.

[289] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 2003.

[290] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Network and Distributed Systems Security (NDSS) Symposium*, 2018.

[291] Curt Schimmel. *UNIX systems for modern architectures: symmetric multiprocessing and caching for kernel programmers*. Addison-Wesley Publishing Co., 1994.

[292] C. P. Schnorr. A Hierarchy of Polynomial Time Lattice Basis Reduction Algorithms. *Theor. Comput. Sci.*, 1987.

[293] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3), January 1991.

[294] Guido Schryen. Security of Open Source and Closed Source Software: An Empirical Comparison of Published Vulnerabilities. *AMCIS 2009 Proceedings*, 2009.

[295] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security Symposium*, 2017.

[296] Michael Schwarz. https://twitter.com/misc0110/status/1129305720770498561, May 2019.

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

[298] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2018.

[299] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2018.

[300] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[301] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *International Conference on Financial Cryptography and Data Security*, 2017.

[302] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2019.

[303] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[304] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 2015.

[305] Atle Selberg. An elementary proof of the prime-number theorem. *Annals of Mathematics*, 1949.

[306] Yannick Seurin. On the exact security of schnorr-type signatures in the random oracle model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2012.

[307] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2004.

[308] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite. *arXiv preprint arXiv:2006.15007*, 2020.

[309] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide Page Deduplication in Virtual Environments. In *international symposium on High-Performance Parallel and Distributed Computing*, 2012.

[310] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. Summarizing known attacks on transport layer security (TLS) and datagram TLS (DTLS). *RFC 7457*, 2015.

[311] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed Systems Security (NDSS) Symposium*, 2017.

[312] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[313] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[314] Signal. Private Contact Discovery Service. https://github.com/signalapp/ContactDiscoveryService.

[315] Rohit Sinha, Sriram Rajamani, and Sanjit A Seshia. A Compiler and Verifier for Page Access Oblivious Computation. Technical report, Technical Report UCB/EECS-2017-124, EECS Department, University of California, Berkeley, 2017.

[316] Igor Skochinsky. Intel ME Secrets. *Code Blue*, 2014.

[317] Michael John Sebastian Smith. *Application-specific integrated circuits*. Addison-Wesley Reading, MA, 1997.

[318] Jonathan Sorenson. An analysis of Lehmer's Euclidean GCD algorithm. In *International symposium on Symbolic and algebraic computation*, 1995.

[319] Evan R Sparks and Evan R Sparks. A security assessment of trusted platform modules computer science technical report TR2007-597. *Dept. Comput. Sci., Dartmouth College, Hanover, NH, USA, Tech. Rep., TR2007-597*, 2007.

[320] ST Microelectronics. CC for IT security evaluation: Trusted Platform Module ST33TPHF2E mode TPM2.0. https://www.ssi.gouv.fr/uploads/2018/10/anssi-cible-cc-2018_41en.pdf, 2019. Accessed: December 10, 2020.

[321] ST Microelectronics. ST33TPHF2ESPI Product Brief. https://www.st.com/resource/en/data_brief/st33tphf2espi.pdf, 2019. Accessed: December 10, 2020.

[322] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480*, 2018.

[323] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[324] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Network and Distributed Systems Security (NDSS) Symposium*, 2016.

[325] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The First Collision for Full SHA-1. In *Annual International Cryptology Conference*, 2017.

[326] Raoul Strackx and Frank Piessens. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks. *arXiv preprint arXiv:1712.08519*, 2017.

[327] strongSwan. Trusted Platform Module 2.0 - strongSwan. https://wiki.strongswan.org/projects/strongswan/wiki/TpmPlugin, 2019. Accessed: December 10, 2020.

[328] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in IaaS clouds. In *Network and Distributed Systems Security (NDSS) Symposium*, 2018.

[329] Atsushi Takayasu and Noboru Kunihiro. Partial Key Exposure Attacks on RSA: Achieving the Boneh-Durfee Bound. In *International Conference on Selected Areas in Cryptography*, 2014.

[330] Andrew S Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 2015.

[331] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX Annual Technical Conference (ATC)*, 2018.

[332] PC TCG. Client Specific-TPM Interface Specification (TIS) Version 1.2. *Trusted Computing Group*, 2005.

[333] PC TCG. TPM 2.0 Mobile Command Response Buffer Interface. *Trusted Computing Group*, 2014.

[334] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 2010.

[335] Philip Frank Terry. Method and apparatus for switching routable frames between disparate media, May 2000. US Patent 6,061,356.

[336] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.4)*, 2019. https://www.sagemath.org.

[337] Daniel Townley and Dmitry Ponomarev. SMT-COP: Defeating Side-Channel Attacks on Execution Units in SMT Processors. In *ACM International Conference on Parallel Architectures and Compilation Techniques*, 2019.

[338] Stephen M Trimberger. *Field-programmable gate array technology*. Springer Science & Business Media, 2012.

[339] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *arXiv:1802.03802*, 2018.

[340] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 2010.

[341] Trusted Computing Group. Protection Profile PC Client Specific TPM. https://trustedcomputinggroup.org/wp-content/uploads/TCG_PP_PCClient_Specific_TPM2.0_v1.1_r1.38.pdf, 2019. Accessed: December 10, 2020.

[342] Trusted Computing Group. TPM 2.0 Library Specification. https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2019. Accessed: December 10, 2020.

[343] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2003.

[344] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture (ISCA)*, 1995.

[345] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[346] Luke Valenta, Nick Sullivan, Antonio Sanso, and Nadia Heninger. In search of CurveSwap: Measuring elliptic curve implementations in the wild. In *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2018.

[347] Jo Van Bulck. Side-Channel Attacks for Privileged Software Adversaries. *PhD Dissertation*, 2020.

[348] 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.

[349] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[350] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[351] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

[352] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[353] 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.

[354] 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 SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[355] Wim Van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 1985.

[356] Stephan van Schaik, Alyssa Milburn, Sebastian Osterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[357] Mathy Vanhoef and Frank Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[358] Vassilios Ververis. Security evaluation of Intel's active management technology, 2010.

[359] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *USENIX Security Symposium*, 2017.

[360] 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 SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[361] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *IACR Cryptol. ePrint Arch.*, 2004.

[362] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[363] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *Proceedings of the ACM on Programming Languages*, 2019.

[364] AF Webster and Stafford E Tavares. On the design of S-boxes. In *Advances in Cryptology*. Springer, 1986.

[365] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *Asia Conference on Computer and Communications Security (AsiaCCS)*, 2018.

[366] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. https://foreshadowattack.eu/foreshadow-NG.pdf, 2018.

[367] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020.

[368] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[369] WikiChip. Ivy Bridge - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/ivy_bridge_(client). Accessed: December 10, 2020.

[370] WikiChip. Kaby Lake - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake. Accessed: December 10, 2020.

[371] WikiChip. Skylake (client) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client). Accessed: December 10, 2020.

[372] WikiChip. Macro-Operation Fusion (MOP Fusion). https://en.wikichip.org/wiki/macro-operation_fusion, 2020.

[373] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[374] WolfSSL. WolfSSL Intel SGX + FIPS 140-2! https://www.wolfssl.com/wolfssl-intel-sgx-fips-140-2/. Accessed: December 10, 2020.

[375] G.M. Wolrich, V. Gopal, K.S. Yap, and W.K. Feghali. SMS4 acceleration processors, methods, systems, and instructions, June 2016. US Patent 9,361,106.

[376] David Wong. Timing and Lattice Attacks on a Remote ECDSA OpenSSL Server: How Practical Are They Really? *IACR Cryptology ePrint Archive*, 2015, 2015.

[377] 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.

[378] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[379] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. vCAT: Dynamic Cache Management using CAT Virtualization. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[380] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[381] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security Symposium*, 2020.

[382] K. Yap, G. Wolrich, S. Satpathy, S. Gulley, V. Gopal, S. Mathew, and W. Feghali. SMS4 acceleration hardware, November 2016. US Patent 9,503,256.

[383] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014.

[384] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[385] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant-time RSA. *Journal of Cryptographic Engineering*, 2017.

[386] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (S&P)*, 2009.

[387] Tatu Ylonen, Chris Lonvick, et al. The Secure Shell (SSH) Protocol Architecture, 2006.

[388] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[389] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. CloudRadar: A Real-time Side-channel Attack Detection System in Clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016.

[390] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2012.

[391] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.