UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Aus dem Institut für IT-Sicherheit
der Universität zu Lübeck
Direktor: Prof. Dr.-Ing. Thomas Eisenbarth

Security and Confidentiality on Shared Computational Resources

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck

Aus der Sektion Informatik / Technik

vorgelegt von
Ida Dorothee Bruhns
aus Hamburg

Lübeck, 2025

1. Berichterstatter: Prof. Dr. Thomas Eisenbarth

2. Berichterstatter: Dr. Maciej Liskiewicz

Tag der mündlichen Prüfung: 29.05.2026

Zum Druck genehmigt. Lübeck, den 01.06.2026

Abstract

The distinction between local and remote computing is increasingly blurred as modern computation relies extensively on the use of shared resources. Pervasive sharing of computational resources is evident in many use cases such as cloud computing, where computational tasks are outsourced to remote servers. Additionally, rented servers, Virtual Private Networks (VPNs), and even web browsers often rely on shared hardware infrastructure.

While the benefits of shared computing resources, such as scalability and cost-effectiveness, are well-documented, this trend also introduces novel security risks. The reliance on shared hardware infrastructure creates opportunities for unauthorized access, data breaches, and other malicious activities.

One very prominent example of sharing both hardware and data are machine learning applications. The use of machine learning applications is rapidly increasing in almost every part of our lives, which includes granting them access to highly sensitive information like health or credit data. At the same time, the models that are used grow larger and larger, necessitating substantial computational resources. This surge in resource consumption has led to a rise in outsourcing both training and inference processes, resulting in the processing of sensitive data on untrusted machines. In this thesis, we examine how to protect data in distributed machine learning systems. In particular, we look at outsourced computations on a machine with a Trusted Execution Environment (TEE) and a fast processing unit, such as a Graphics Processing Unit (GPU). I examined the SLALOM protocol, a seminal work in privacy-preserving inference. In this theses I present a new method, CARNIVAL, to significantly speed up the preprocessing phase. CARNIVAL leverages the pseudo-randomness of the *Subset sum problem* to enable efficient outsourcing during the preprocessing phase. The findings from the performance benchmarks demonstrate that CARNIVAL is a promising candidate for real-world implementations. A second possibility to continue working with the SLALOM framework, DASH, is introduced briefly. It builds on arithmetic Garbled Circuits (GCs) in combination with a TEE.

However, TEEs do not provide protection against microarchitectural side channel attacks. With the SPOILER technique, we will see how *speculative loads* and *store forwarding*, a previously undocumented behavior of Intel processors, can be exploited to leak information. Specifically, I show that the processor's dependency prediction mechanism can be used to unveil information about the physical page mapping, which is a novel side channel that can be used to boost Rowhammer-style attacks or find eviction sets for cache attacks.

As microarchitectural side channel attacks have been so prominent in security research over the last few years, countermeasures were also an active research field. Consequently, all major cryptographic libraries provide countermeasures to hinder key extraction via cross-core cache attacks by now. In the last part of this thesis, we analyze prefetch-based countermeasures aimed at preventing well-known cache attacks. I demonstrate that these implementations remain vulnerable under certain conditions and propose a novel attack technique called CACHESNIPER that enables the attacking process to evict the target data from the cache at the desired instants after the prefetch but before the utilization. CACHESNIPER leverages transient capabilities provided by Intel TSX and a deep knowledge of the L3 cache replacement policy. It is applicable by an unprivileged attacker.

This thesis makes several significant contributions to the field of secure outsourced computation, including the development of a novel method for protecting sensitive data during inference, the discovery of a previously unknown microarchitectural side channel that can be exploited to enhance existing attacks, and a thorough examination of the existing countermeasure of prefetching, including its limitations and vulnerabilities which in turn led to a novel side channel attack.

Kurzfassung

Die Unterscheidung zwischen lokalem und Remote-Datenverarbeitung wird zunehmend verwischt, da moderne Anwendungen stark auf die Nutzung gemeinsamer Ressourcen angewiesen sind. Die weitverbreitete Nutzung gemeinsamer Rechenressourcen zeigt sich in vielen Anwendungsfällen wie Cloud-Computing, bei dem Berechnungen an entfernte Server ausgelagert werden. Darüber basieren angemietete Server, virtuelle Private Netzwerke (VPNs) und sogar Webbrowser häufig auf gemeinsamee Hardware-Infrastruktur.

Während die Vorteile gemeinsamer Rechenressourcen, wie Skalierbarkeit und Kosteneffizienz, gut dokumentiert sind, birgt diese Entwicklung auch neue Sicherheitsrisiken. Die Abhängigkeit von gemeinsamer Hardware-Infrastruktur schafft Möglichkeiten für unautorisierten Zugriff, Datenschutzverstöße und andere Angriffe.

Ein sehr prominentes Beispiel für die gemeinsame Nutzung von Hardware und Daten sind Anwendungen für maschinelles Lernen. Die Verwendung von diesen Anwendungen nimmt in fast allen Bereichen unseres Lebens rapide zu, was auch bedeutet, dass ihnen Zugriff auf sehr sensible Informationen wie Gesundheits- oder Kreditdaten gewährt wird. Gleichzeitig wachsen die Modelle, die verwendet werden, immer weiter an, was erhebliche Rechenressourcen erfordert. Diese Steigerung der benötigten Ressourcen hat zu einer Zunahme der Outsourcings von Trainings- und Inferenz-Prozessen geführt, was wiederum zu einer Verarbeitung sensibler Daten auf unvertrauenswürdigen Maschinen führt. In dieser Arbeit untersuchen wir, wie man Daten in verteilten maschinellen Lernsystemen schützen kann. Insbesondere betrachten wir ausgelagerte Berechnungen auf einem Computer mit einer vertrauenswürdigen Ausführungsumgebung (Trusted Execution Environment (TEE)) und einem schnellen Prozessoreinheit, wie einer Grafikkarte (Graphics Processing Unit (GPU)). Ich habe das SLALOM-Protokoll, ein bahnbrechendes Werk im Bereich der datenschutzfreundlichen Inferenz. In dieser Arbeit stelle ich eine neue Methode, CARNIVAL, vor, die es ermöglicht, die Vorverarbeitungsphase

von SLALOM erheblich zu beschleunigen. CARNIVAL nutzt die Pseudozufälligkeit des *Teilsummenproblems*, um eine effiziente Auslagerung während der Vorbereitungsphase zu ermöglichen. Die Ergebnisse der Performance-Messung zeigen, dass CARNIVAL ein vielversprechender Kandidat für reale Implementierungen ist. Eine zweite Möglichkeit, mit dem SLALOM-Framework weiterzuarbeiten, DASH, wird kurz vorgestellt. DASH baut auf arithmetischen Garbled Circuits in Kombination mit einem TEE auf.

TEEs bieten jedoch keinen Schutz gegen mikroarchitektonische Seitenkanalangriffe. Mit der SPOILER Technik werden wir sehen, wie *speculative loads* und *store forwarding*, ein bisher nicht dokumentiertes Verhalten von Intel-Prozessoren, ausgenutzt werden können, um Informationen zu gewinnen. Insbesondere zeige ich, dass der der Vorhersagealgorithmus für Abhängigkeiten von Speicheradressen des Prozessors dazu verwendet werden kann, Informationen über physikalische Speicheradressen zu enthüllen. Dies ist ein neuartiger Seitenkanal, der dazu verwendet werden kann, Rowhammer-ähnliche Angriffe zu verstärken oder Eviction Sets für Cache-Angriffe zu finden.

Da Mikroarchitektur-Seitenkanalangriffe in den letzten Jahren in der Sicherheitsforschung eine so große Rolle gespielt haben, waren auch Gegenmaßnahmen ein aktives Forschungsgebiet. Infolgedessen bieten mittlerweile alle großen kryptographischen Bibliotheken Gegenmaßnahmen an, um die Schlüsselextraktion durch Cross-Core-Cache-Angriffe zu verhindern. Im letzten Teil dieser Arbeit analysieren wir Prefetch-basierte Gegenmaßnahmen, die darauf abzielen, bekannte Cache-Angriffe zu verhindern. Ich zeige, dass diese Implementierungen unter bestimmten Bedingungen anfällig bleiben, und stelle eine neuartige Angriffstechnik namens CACHESNIPER vor. CACHESNIPER ermöglicht es dem angreifenden Prozess, die Zieldaten zu einem bestimmten Zeitpunkt, nämlich nach dem Prefetch, aber vor der Nutzung, aus dem Cache zu verdrängen. CACHESNIPER nutzt die von TSX bereitgestellten transienten Fähigkeiten sowie genaue Kenntnis der L3-Cache Replacement Policy. Er kann von einem Angreifer mit normalen Nutzerrechten angewendet werden.

Diese Arbeit leistet mehrere wichtige Beiträge im Bereich der ausgelagerten Datenverarbeitung, einschließlich der Entwicklung einer neuartigen Methode zum Schutz sensibler Daten während der Inferenz, der Entdeckung eines bisher unbekannten mikroarchitektonischen Seitenkanals, der zur Verbesserung bestehender Angriffe ausgenutzt werden kann, und einer gründlichen Untersuchung der bestehen-

den Gegenmaßnahme des Prefetching, einschließlich ihrer Einschränkungen und Schwachstellen, die wiederum zu einem neuartigen Seitenkanalangriff führten.

Acknowledgements

When asked why I wanted to do a PhD I always answered "It’s my pony". All other little girls dreamed of having a pony, and I dreamed of being a scientist (at the time marine biologist) and having a PhD. It turned out that analogy goes very far: There is a lot of work, very little carefree time and the thing never seems to go where you want it to. But then something happens: you meet other pony owners with similar problems. Some of them are nice, and some have already tackled the problems you are facing. Somebody helps you, or you find out that carrots just work better than apples on your specific pony. And in a way, it works, but not as magically as you hoped.

The Universität zu Lübeck has been my home for almost twenty years now. As with any good home, I left it, and I was welcomed back. It is a great place, with great people. If anyone wants to visit or work there, I strongly recommend it.

I thank Thomas Eisenbarth for giving me a chance when I was not a traditional PhD candidate. Thank you for the deep insight into your field of expertise, the discussion of ideas and helpful pointers when I needed them. We met when you were just starting out in Lübeck. You built something very amazing here. I don’t think there was a research group that grew so fast at our university before. I would have liked to start working in it now, and enjoy the dynamic. I wish you and everyone in your team the very best.

To Sebastian Berndt, you are an amazing person. Leading people in a selfless way, being there if needed and very positive in every situation. I am very glad that you defined listening as in-scope of your tasks as a post doc. Working with you was a pleasure, and I would be glad if we could continue our lunches.

My lovely office mates and then friends, Okan Seker and Thore Tiemann: thank you for being there and not judging too much. Thank you for listening, for sharing chocolate as well as ideas and worries. Thank you my strange smelling tea did not bother you.

I met a lot of amazing people on the way, but since you cannot escape feminism as a female computer scientist, I will emphasize the women: Irem, I hope you will be a cryptographer. It's a great career choice, and hearing you utter it with conviction made me light up inside. Vasha, at some point we will put our two big brains together. And it will be just as fun as having conversations in weird places. Stay fearless. Susanne, thank you for calling me brave when I felt as small as possible. Veelasha, always a pleasure. You never make me feel weird for reaching out.

Johanna Ahnsohn-McDougal, I never would have met you without this thesis. You are fierce and brave and wonderful. And you always seem to think I am the same or similar things, which amazes and encourages me. Thank you for your friendship, guidance and honesty. I value our friendship, and if you are willing, we can trade apple rings for melon slices until we are 80.

Samira Briongos, this thesis would not have happened without you. You are the only colleague who ever waited for me with their research (what were you thinking?), and the only one available at two in the morning. With endless humor, forgiveness and the constant threat of burpees, you helped me a lot. If you ever need anything ever, please find me. Until then, let's stay friends.

My lovely Daniela Freitag, it's funny that you kept me from going insane. Thanks for long walks, many liters of tea and snacks a children's birthday party could only dream of. And for looking at weird scary diagrams explaining my research and career situation.

To my parents and my family. I always knew you would be there when I needed you. That is a strength and a superpower and a secret, all in one.

Lastly, the three remaining members of "our quartet": Erik, you make me see the world in weird and wonderful ways. If you were my only task it would be worth it. Thank you for your worry, and your care in form of coffee, tea and blankets. Linea, I never knew a person who's appreciation meant more to me. Thank you for your sunny moods, for the feminist adoration and everything that you are. Showing you what's possible is my best motivation. Martin, I love you. Thank you for making everything work that I needed. Thank you for helping me forgive myself. Thank you for all the laughs. You told me that everything will be very, very well. And I believe it. Forever.

In the end, I got my pony. And despite not knowing if it was worth the trouble, I kind of like it.

Contents

List of Figures

List of Abbreviations

CPU Central Processing Unit. 4, 5, 17, 18, 24, 25, 37, 95, 98

CVE Common Vulnerability Enumeration. 7, 8, 65, 81, 87, 110

DRAM Dynamic Random Access Memory. 11, 20, 21, 23, 24, 27

FPGA Field-programmable Gate Array. 13, 19, 38

FPU Fast Processing Unit. 4–6, 37–39, 41–44, 48, 52–54, 56–58, 60, 63, 64

GC Garbled Circuit. v, 16, 17, 58, 60–62, 64

GPU Graphics Processing Unit. v, vii, 5, 6, 11, 13, 17, 18, 35, 38, 54, 55, 58–62, 64

HE Homomorphic Encryption. 16, 61, 62

HPC Hardware Performance Counter. 68

IaaS Inference as a Service. 14, 15, 37, 52

LLC Last Level Cache. 4, 27, 88, 92, 98, 101, 102, 106, 115, 117

LRU Least Recently Used. 30

ML Machine Learning. 2, 11, 13, 17, 36, 54, 60, 64, 119

MLaaS Machine Learning as a Service. 2, 14, 15, 19, 37

MMU Memory Management Unit. 23, 24, 29

MOB Memory Order Buffer. 24, 25, 66, 69

MRU Most Recently Used. 30

NN Neural Network. 2, 11, 41

NPU Neural Processing Unit. 13, 18, 19

OTP One-time Pad. 39–41, 47, 48, 54, 61

SAB Store Address Buffer. 24

SDB Store Data Buffer. 24

SMPC Secure Multi-party Computation. 16, 17, 61, 62

TEE Trusted Execution Environment. v–viii, 4–6, 8, 11, 17, 19, 20, 35, 37–39, 41–44, 52–61, 63–65, 78, 119–121

TLB Translation Look-aside Buffer. 23, 29, 66, 80, 81, 83

TPU Tensor Processing Unit. 13

VM Virtual Machine. 19, 22

VPN Virtual Private Network. v

1

Introduction

Once upon a time, in a country far far away, a lady named Alice sits in front of her computer. Alice wants to apply for a loan for a house. She enters all her personal and credit information into her bank's software, hits *send*. The bank software processes the data while Alice anxiously waits for a result.

At the same time, but in a totally different place, a man named Bob just found a great new web service that allows him to do photo and video editing with the latest software. Bob pays a small fee and uploads his latest personal photos and videos. They show him and his friends, in funny and also mildly embarrassing situations. Bob is amazed by the website, it is reasonably priced, offers the latest software and also processes his edits very fast.

Alice and Bob have never met. They never talked. They don't know it, but they still have something in common: They are actually working on the same computer. Alice's bank just happens to use an artificial neural network to check people's credit score that runs on an external server. The same server runs Bob's photo editing software. Both the bank and Bob use the server as a service provider, once for inference on a neural net, and once for running software. The bank does not need its own machine learning server, as it only requires it for loan applications. Bob edits his photos as a hobby, he is very happy he does not need to invest in a new PC and software but can just start working on the remote PC for a small fee.

In modern day computing, most computations are not happening on a private machine, but on shared hardware. There are many reasons for this. Web services, for instance, are offered centrally since a decentralized architecture is typically neither required nor sensible. Additionally, not every company or person wants to or can support their own infrastructure for every task. External servers offer many advantages such as availability guarantees and monitoring while at the same time eliminating the need for administrative personnel in the own company, providing scalability to different customer demands, or simply not having to provide the space

for the machine. More powerful hardware or specialized equipment can be used for a limited time frame instead of being bought, knowledge and data can be shared. Sharing hardware thus makes sense in an economical, ecological and quality sense.

One of the most resource intensive applications nowadays revolves around the field of *Machine Learning (ML)*. It has been in the scientific and economic spotlight for over a decade now and deep neural networks are known to require a large amount of computing power. The two most common tasks here are the *training* of a Neural Network (NN) and the *inference* or evaluation of such a trained network.

Machine learning is a good example for shared resources in several ways: First of all, the machine load comes in irregular intervals. Especially the training of a network requires extensive amounts of computational and storage resources over a relatively short amount of time when compared to the total lifetime of the neural network, which can well be answered by shared hardware. Let it be noted that this relatively short time may still be weeks or months for large models. Secondly, training requires vast amount of data, possibly more or more varied than a single party can provide. Thirdly, models are usually trained by one party and then used by others. Lastly, inference tasks for the trained network may not come in at a steady flow but in bursts. Thus, the model may be hosted on shared hardware to either handle fluctuations in demand or simply to provide a service to customers.

As a result, *Machine Learning as a Service (MLaaS)* is a booming field: Smaller parties can leverage external hardware to train models, utilize shared data or infere on pre-trained models. This enables them to participate in machine learning without requiring significant computational resources or data storage [55]. However, outsourcing computation may raise ethical or legal issues depending on the processed data. It is easy to see why data sets such as health data, credit statements, or a company's intellectual property are not suitable to be processed on a shared, insecure machine without further protection. Both privacy preserving training and inference are active research fields, working to protect the training and inference data as well as the model and the output [55, 59, 142, 169, 239].

Let us zoom out from the use case of machine learning. While shared hardware brings many advantages with respect to resource consumption, it also introduces a variety of new challenges in the field of privacy and security. After all, shared hardware is used for sensitive operations such as banking, encryption and decryption, processing of medical data, streaming of private videos, control of critical systems,

and many others. As the user is not the only user on the machine, new attack channels open for both co-tenants and the hardware owner. It is easy to see that three common security goals may be compromised:

Confidentiality The data sent to the shared machine could be sold, or leaked.

Integrity The program code could be changed. An attacker could tamper with input or output.

Availability The business model of offering hardware or remote services is vulnerable to DOS attacks, endangering the availability of the service.

One attack class that is unique to shared hardware are *microarchitectural side channel attacks*: On the shared machine, the microarchitecture is shared by all processes. That means that resources like the cache or the address resolution logic are used by processes of different tenants at the same time. By carefully preparing and monitoring the shared resources, an attacker in one process can infer information about a victim in the other process. Many research groups showed a plethora of ways to extract different types of private information, starting from browsing patterns and visited websites down to cryptographic keys [26, 68, 121, 173].

Ultimately, the shared use of hardware requires a careful balance between the benefits and risks. As mentioned in the introduction, end users are often not even aware that they are using shared hardware. It is crucial for manufacturers and operators of shared hardware systems to implement robust security measures to protect their customers' privacy and security. Privacy and security on shared hardware thus offers a very diverse research field.

1.1 Setting

As mentioned above, Alice and Bob are both sitting in front of their own device, but are not actually working on it. Bob is using a software as a service over the internet, working on a remote server he does not know anything about. Alice is entering her information into her bank's form, which is then entered into a MLaaS application also running on the same remote server.

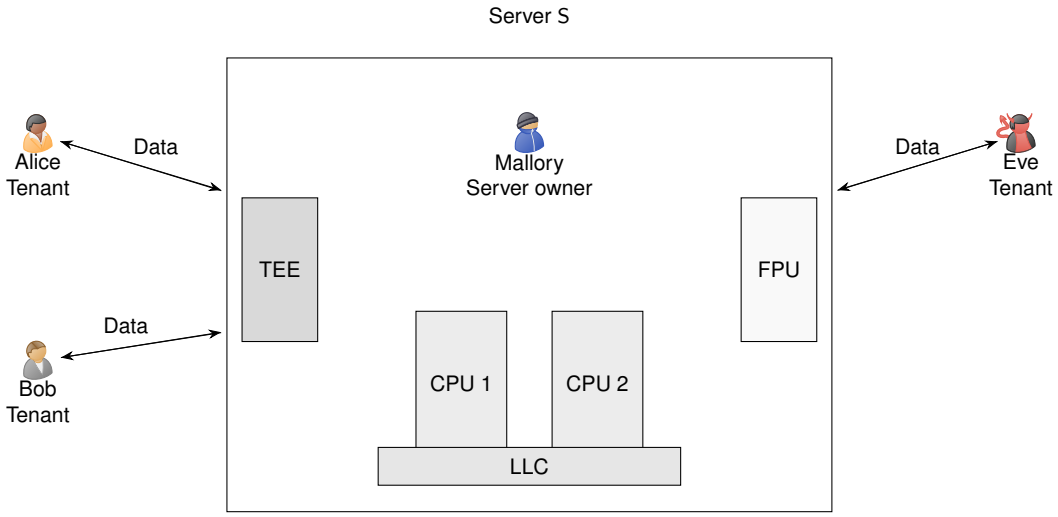

Figure 1.1: Basic attack scenario: Mallory hosts a server S, Alice and Bob are honest tenants, Eve is an attacking tenant. The server contains several CPUs that share a Last Level Cache (LLC). The server may contain a Trusted Execution Environment (TEE) and a Fast Processing Unit (FPU). In the chapters, we omit components that are not essential to the respective chapter.

We can put this scene into a more abstract setting. We assume that the shared hardware, called the server S in the following, can be accessed by several parties at once. The scenario is depicted in Figure 1.1: S belongs to Mallory. Alice, Bob and Eve are tenants on the server. They all have user-level privileges, while Mallory has root-level privileges as well as physical access to S.

As customary in cryptography, Alice and Bob are honest parties [187, 198]. Mallory and Eve are trying to attack them and their calculations.

They take one of the following classes for each scenario [71]:

Honest-but-curious / Semi-honest The attacker sticks to the agreed upon protocols, but will access any information they can within the protocol. For example, she will access data that is unencrypted, save messages for later analysis or evaluate side channels. The attacker may also provide chosen inputs to the protocol.

Malicious The attacker deviates from the protocol and uses whatever means necessary to gain information. The deviations may include fault attacks, replay, deletion, fiddling with random number generators and many more.

We will discuss the attacker type in each part of this thesis individually. They are relevant for our new framework for outsourced computation. Side channel attacks are outside the scope of the attacker model: They are based on vulnerable implementations, but the attacker is strictly speaking not taking part in the protocol.

The server S has multiple Central Processing Unit (CPUs), each with a private Level 1 and Level 2 cache and a shared Level 3 cache. It further contains a TEE to allow clients to perform protected computations. Another important component of S is a Fast Processing Unit (FPU), which is a term we will use for any specialized processing unit faster than the CPU. The communication between the client and the servers is assumed as encrypted with a secure standard protocol such as TLS. The exact components and capabilities of the server as well as the involved parties will be specified as needed during the thesis.

1.2 Research Objectives and Contributions

Within the setting described above, we looked at both a security enhancing framework and potential threats via side channels. This thesis will focus on three research objectives.

1.2.1 RO1: The development of privacy-enhancing techniques for outsourced computation

To narrow down the scope of my research, I focused on the case of outsourced inference: a *model owner* provides a trained model and a *client* provides an input and receives the results. In outsourced inference, there is no inherent protection for either. I mainly focus on the protection of the client data, called *input privacy* and *output privacy*, but also touch on the *integrity of the calculation*. All these are security goals of the client sending data, not the model owner.

I used the SLALOM framework from Tramer and Boneh as a baseline for my research. It was published in 2018 and proposes a way to split a machine learning inference workload between a Trusted Execution Environment (TEE) and a Graphics Processing Unit (GPU) to enhance integrity of the calculation and the privacy of the inputs and outputs. It relies on the TEE and masking for their privacy guarantees.

Unfortunately, SLALOM has a resource intensive preprocessing phase solely on the TEE. We stay in the same setting, but design a new preprocessing phase that makes better use of resources. My new algorithm CARNIVAL generates many masks in public on the GPU. We then make use of the *Subset sum problem over finite fields* to calculate private masks within the TEE.

Contributions I develop a secure method to perform pre-calculations on untrusted hardware. CARNIVAL is a way to use publicly performed calculations to generate secret masks for confidential input values. I prove that CARNIVAL is secure under the assumption that the Subset sum problem over finite fields is a one-way function, building on a longstanding cryptographical assumption. Additionally, I discuss a variant with reduced key size but similar security. We then integrate CARNIVAL into SLALOM and thus propose SLALOM AT THE CARNIVAL (S@C), a new variant of SLALOM with an improved preprocessing phase and an analysis of the performance gain. Even when choosing conservative parameters, this already generates a speedup between 2.2 and 11 in the provable scenario. My approach makes better use of the FPU by putting it to work during the setup phase as well as the online phase. While I demonstrate it with the SLALOM use case, the CARNIVAL primitive can be applied in many settings. I also worked on a second new approach on outsourcing computation by leveraging garbled circuits. It is called DASH and is discussed along with the state of the art of outsourced computation and the related work.

1.2.2 RO2: Analysis and quantification of microarchitectural behavior that allows an attacker to improve on existing attacks or develop new attacks

The security of both CARNIVAL and DASH relies on the security of the TEE. I assume that we have a secure part in the system that provably cannot be compromised by an attacker. While this assumption holds, peculiarities in the implementation can still allow an attacker to infer secrets, even from a TEE. We will look deeper into one attack category, namely *microarchitectural side channel attacks*. In our setting, processes share resources such as the higher level caches, various buffers, the page table, the GPU or just the physical memory. It is possible to infer details about data or the program state of a victim process by monitoring the shared microarchitectural state. As all security research, side channel research is also a

cat-and-mouse-game: Attackers develop new attacks and circumvent existing defenses. Researchers develop detailed knowledge of the microarchitecture, leaking components and sensitive algorithms. They also develop attacks so the defenses can stay ahead of potential attackers. Only after an attack has been fully understood an effective countermeasure be can developed.

One of the shared microarchitectural components is the *store buffer* as well as components for *address resolution* such as the *memory management unit*. I analyzed the store buffer and discovered that issuing many *store* instructions, subsequent *load* instructions are executed *speculatively* without resolving the address fully.

Contributions We discovered and analyzed a novel microarchitectural leakage from the store buffer stemming from the false dependency hazards during speculative load operations: If a potential dependency between the addresses of the load and the buffered *store* instructions is discovered, a significant delay in executing the *load* instructions can be observed. The delay leaks partial physical address information. The new leakage is dubbed SPOILER. It is a novel microarchitectural leakage which reveals critical information about physical page mappings to unprivileged processes. Since the physical address is information typically not available to user space processes, this is a significant security risk. I use SPOILER to speed up the reverse engineering of virtual-to-physical mappings by a factor of 256. The speedup can be observed in native environments, from within virtual machines and sandboxed environments such as JavaScript environments in browsers. I used SPOILER to design a novel eviction set search technique from JavaScript and to generate efficient DRAM row conflicts. As SPOILER allows the attacker to detect continuous physical memory, the first *double-sided Rowhammer* attack with normal user-level privileges could be conducted. After going through the responsible disclosure process, SPOILER received a CVE from Intel.

1.2.3 RO3: Analysis of existing countermeasures to microarchitectural attacks and privacy-enhancing mechanisms and their improvement

As mentioned in Subsection 1.2.2, researchers propose countermeasures against side channel attacks. Evaluating how well these countermeasures work and whether they can be circumnavigated is also a research objective. One countermeasure

against attacks that infer secrets from the cache state is to avoid leaking cache behavior by always fetching all data into the cache, independent of its actual utilization. These are called prefetch or always-load strategies.

I craft an attack to retrieve secrets from protected implementations. The attacker needs to solve four challenges: The attacker needs to be able to *detect* when the victim is running the target algorithm. She has to *determine the time window* between the detection of the target algorithm and the utilization of the target data. At exactly the right instant, she needs to *evict* the target data from memory. As in previous attacks, she then needs to *recover* the information about a potential access of the victim algorithm. I analyze different methods to tackle these challenges and evaluate which work best by testing with a synthetic benchmark. With Intel TSX, I craft an elegant attack, CACHESNIPER.

Contributions I show that a classic user-level cache adversary can overcome a prefetch protection by solving the four key challenges stated above. We use the result to craft CACHESNIPER, a novel attack technique that allows a user level attacker to leverage tiny windows of opportunity. It achieves high precision by combining a TSX transaction to precisely determine the victim process's state, the corresponding abort handler to directly conduct the attack, and fast and accurate eviction techniques to get data from the cache. CACHESNIPER was used to retrieve keys from protected AES and RSA implementations, both from real world libraries. revealing that last-level cache attacks against protected implementations are still possible even without special privileges. After going through a responsible disclosure process, CACHESNIPER received a CVE for the attack on the RSA implementation of WolfSSL.

1.3 Structure of this Thesis

The background information for this thesis is presented in Chapter 2. It will introduce all concepts required to read the remainder of the thesis. Chapter 3 starts with the baseline of SLALOM, a framework for private outsourced inference from 2018. From there, I develop the CARNIVAL protocol, which uses the Subset Sum problem to fix several weaknesses of SLALOM. I use it to generate the new framework SLALOM *at the* CARNIVAL (S@C). As an alternative, we discuss DASH, a framework building on garbled circuits instead of a TEE.

Chapter 4 introduces the SPOILER leakage and analyses its root cause. I provide a detailed description of the new eviction set creation algorithm, which is implemented in JavaScript and thus usable in browser-based attacks. SPOILER can be used to improve other attacks like Rowhammer style attacks, which I also describe. The chapter includes mitigations and countermeasures.

After that, Chapter 5 discusses the possibility of attacking prefetch-protected implementations and introduces CACHESNIPER as an attack that can do so. In addition to two attacks on real-world libraries I also present possible countermeasures. Chapters 3 through 5 each contain a description of the work distribution in the corresponding publications as well as the respective related work. The thesis concludes in Chapter 6 by revisiting the research objectives, offering ideas for future work and giving closing remarks.

2

Background

This chapter first provides a short overview on machine learning, private inference and secure outsourced computations. We then dive into computer architecture, starting on a high level with Trusted Execution Environments (TEEs) and specialized processing units such as Graphics Processing Units (GPUs). Process isolation, hypervisors and address virtualization are discussed next before a deep dive into individual microarchitectural components. Of these, the store buffer, the DRAM and the cache, including some known attacks, will be discussed.

2.1 Machine Learning on a Very High Level

Machine Learning (ML) is a booming field across all industries. It is used to suggest products in online shopping, help cars drive, order images, generate text, chat with users, classify test results, detect diseases, and many more. As it works with large amounts of possibly private data on possibly private models, we use it as a use case for secure outsourced computation.

Very simplified, machine learning is used to build a *Neural Network (NN)*. The model is built during a process called *training*. Once training is complete, the NN or model can *infer* on previously unseen input data.

A network consists of multiple *layers*, which can be either linear or non-linear. We use the VGG-16 net from Simonyan and Zisserman as an example [203]. It is shown in Figure 2.1.

The VGG-16 network is used for image classification. It uses the ImageNet dataset[1] and was a submission to the ImageNet classification challenge in 2014. It is a fairly small network with a uniform structure, which makes it popular for research

[1] `https://www.image-net.org/`

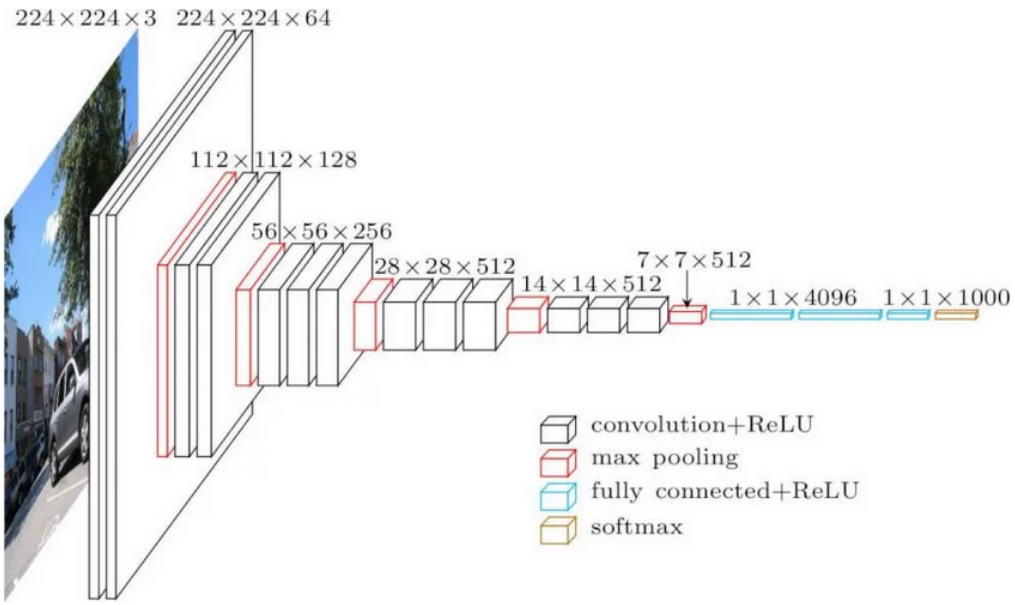

Figure 2.1: The architecture of the VGG-16 network. Is is used for image classification. Figure from [86]

and teaching. VGG-16 takes image files of 224×224 pixels in RGB format as input. These are classified through 13 convolutional layers directly followed by a ReLU activation function, each using 3×3 kernels. Convolutional layers and fully connected layers are linear layers, while max pooling layers, ReLU layers and softmax layers are non-linear layers.

The switch between linear and non-linear layers is the main difficulty when attempting secure outsourced ML: linear functions can be secured efficiently with homomorphic encryption or masking, while the non-linear functions can be secured efficiently with garbled circuits. It is, however, vastly computationally expensive to switch between these two methods. Some efforts go into approximating activation functions with functions that can be calculated homomorphically, such as polynomials, but so far the computational trade-off is too large and the accuracy of the network suffers [66, 152]. While accuracy can be re-improved by co-designing the network (e.g. by retraining it) to match the activation functions, we can take away that approximated activation functions usually do not match out-of-the-box networks [55].

Machine learning applications often leverage specialized hardware. As machine

learning is a classic example of single-instruction-multiple-data, it can be very easily adapted to Graphics Processing Units (GPUs). Moving to the GPU is a natural step as one of the most common linear layers, convolution, is a core calculation on GPUs anyway. However, more specialized hardware was developed as well: There are examples of neural nets in Field-programmable Gate Arrays (FPGAs) [115, 133]. Recently, many consumer devices are additionally equipped with a Neural Processing Unit (NPU), hardware units built entirely for the purpose of running ML applications. Among others, Google uses NPUs in StreetView and search result ranking [148]. Lastly, Tensor Processing Unit (TPUs) are a hardware type coined by Google that fulfill the same tasks as Neural Processing Unit (NPUs), but are designed to work in large clusters as opposed to being part of consumer devices [122].

2.1.1 Training

For training, also called learning, a model requires training data. A data set is split into training data and test data. A first version of the network is built: it has all necessary layers, but the parameters within the layers are dummy values. For example, a convolutional layer is defined, but the values within the filter are random. Often, a known network architecture for the given use case is used and trained on the specific data [73]. For example, the VGG16 net is designed for image classification in general images, and may thus be adapted to assist in evaluating medical images.

The training data is sent through the network in a process called *forward propagation*. By comparing the expected and actual results, an error term is determined. The network parameters are then adjusted to minimize the error in a process called *backward propagation*. The process of forward- and backward propagation is repeated many times, until the training error falls below a given threshold or further training does not reduce the error term further. Then the test data set is used to test how well the network performs. It can then be retrained or used for inference.

Neural networks typically have a huge number of parameters, which in turn means tuning them requires large amounts of data. These need to be collected or generated and pre-processed, for example by bringing them to a standard size or labeling them. Sharing data in a secure way would thus allow more parties to contribute to the task of data gathering. Then, many rounds of forward and backward propagation

take place, each round consisting of classification, error calculation and parameter correction in a multidimensional plane. Thus in addition to divers, preprocessed data, learning requires massive amounts of computing power. After the learning phase is complete, the requirements for computing power are reduced drastically. Consequently, using external machines securely for learning is desirable.

However, there are inherent problems with secure outsourced learning. Both the input data and the model parameters may be secret. The model parameters are also constantly changing during training, which makes it difficult to pre-compute anything. Most adjustments through propagation do not work on masked values [213].

A note on the economic aspect: both data and trained models are valuable assets that warrant protection. The value lies in the substantial time and computational resources required to collect, preprocess, and train models. Data can be acquired through purchase, with a significant market existing for this purpose. Similarly, models can be licensed or sold, giving rise to the burgeoning field of MLaaS. This economic sector is characterized by the provision of trained models and associated expertise, enabling organizations to leverage the benefits of machine learning without incurring the costs of developing and maintaining their own models [180].

2.1.2 Inference

Assuming we managed to train a model to our satisfaction, we can now use it to classify previously unknown data. As not everybody has their own trained model for every task, readily trained models are an asset. Users can now send their data to an inference server and receive the classification. Inferring a model can happen consciously or unconsciously: For example, many mobile phones with online data storage offer an automatic classification for the user's photos. The model owner can include the classification as an explicit service or, as described above, implicitly as part of another service.

Inference is much less computation intensive than training. However, there are still many reasons to use Inference as a Service (IaaS):

- The responsibility for the availability of the service is shifted to the service provider.
- The service provider also needs to tend to scalability.

- The service provider owns and maintains the hardware.
- The model may be updated regularly with current data.
- A trained model still needs to be stored and used, and models are often very large with thousands of parameters.
- The machine learning expertise can stay with the service provider.

In research, IaaS solutions are for example used in nuclear physics [58, 222].

The data sent to the model owner can be of different privacy value: inferring traffic camera images or interpreting speech orders to Alexa may be fairly unspectacular, but detecting cancer in MRI images or calculating a credit score is much more sensitive. To address this, there are research efforts to automatically classify the privacy level required for specific data, for example photographs shared on social networks [205].

2.2 Secure Outsourced Computation of Machine Learning Tasks

Outsourcing computations to shared machines is a practical and sensible way to use resources: not every party can (and should) have the resources for any necessary computation on premise, maintaining a lot of different specialized hardware is not practical and efficient, and using optimized implementations on shared machines is beneficial. In the field of machine learning, models are usually trained by one party and then used by others, and classification tasks may not come in at a steady flow but in bursts. We can see that it makes sense to outsource the classification tasks to a MLaaS server holding the model, thus providing IaaS. But an IaaS scenario offers no inherent protection of the input values provided by the clients from the service provider or other clients. There is also no built-in protection for the output values. There are three main possibilities to add this protection, all of which come with a performance penalty and sometimes also effect accuracy.

Homomorphic Encryption (HE) Certain types of computations can be performed directly on encrypted data, known as *homomorphic encryption*. One or more parties provide inputs, which are encrypted and sent to a server or untrusted party. The server then performs the computation on the encrypted data, and the resulting ciphertext is sent back to the input providers, who can then decrypt it. Examples of partially homomorphic encryption schemes include RSA, additive masking, and El-Gamal [158]. Fully homomorphic systems need to support a set of Turing-complete set of operations, for example allowing both unlimited multiplications and additions to perform arbitrary calculations. Fully homomorphic systems are currently very slow, and it is unsure if there will ever be a scalable version that can be widely deployed [60].

As linear network layers such as convolutional or fully connected layers can be seen as a series of multiplications by a constant and additions, which are homomorphic operations, they can be calculated with relatively little overhead [46, 69]. The non-linear activation layers like softmax or ReLu layers however are not easy to calculate using homomorphic encryption. They are usually approximated with low-degree polynomials [42, 89] or adapted cosine functions [224] in fully homomorphic approaches. Both approximations come at an accuracy loss and sometimes require extensive retraining of the network. The scheme of choice for machine learning is currently the Cheon-Kim-Kim-Song (CKKS) scheme, which works on polynomial quotient rings and supports batch evaluation [42, 168]. The inaccuracy caused by the approximated activation functions grows with the network depth: The largest neural network evaluated using purely HE on the CIFAR-10 dataset is a net with 10 convolutional layers [168]. The CIFAR-10 dataset is a set of images of hand-written numbers from 0 to 9. The images are 32×32 pixels and thus not very large data structures.

Secure Multi-party Computation (SMPC) Secure Multi-party Computation (SMPC) is a subfield of cryptography that allows mutually untrusted parties to compute a function together without revealing their inputs to each other. As opposed to classic cryptography, SMPC assumes an attacker within the communication instead of an outsider attacking the communication between two honest parties. The data owners also don't have access to a trusted third party.

In 1982, the general idea of computing a function $f(x_1, x_2, \ldots x_n)$ for n input owners was introduced be Yao [235]. Later, Yao formulated the GC protocol. The GC protocol remains the basis for modern SMPC schemes. A *garbler* encrypts the

entries of a truth table representing a function, and sends the ciphertexts and the key for her input to the *evaluator*. The evaluator receives the key for her input bit by oblivious transfer and can decode only the correct entry. Many optimizations like point-and-permute, free `xor` or wire coloring have been introduced. The procedure of encryption and oblivious transfer is repeated successively from the input to the output gates. In the early 2000s, the optimizations led to the first practical applications of SMPC [60].

Many schemes for secure ML work with GCs to encrypt the non-linear layers of networks, or the entire computation. As computing arithmetic operations via traditional binary GCs is very resource intensive, Ball et al. introduced so-called garbling gadgets for efficient garbling of arithmetic circuits over a finite field in 2016 [13]. In their follow up work in 2016, Ball et al. used these gadgets to build neural networks [12].

Trusted Hardware As mentioned above, TEEs are a viable tool to protect privacy on shared hardware. The main drawbacks are the need for specialized hardware and resource limitations e.g. through paging mechanisms. The code within the enclave is obviously victim to any vulnerabilities in the enclave code or concept itself. Another interesting problem is the verification of the attested code: Depending on the problem at hand, the client may not be able to fully verify the code within the enclave. The issue can be circumvented by passing out intermediate results and verifying them on a random sample basis.

Some approaches use a combination of the above, trying to overcome the challenges of one approach by using another. For example, CrypTFlow is a TEE-MPC hybrid solution [125]. Indeed, the masking operation in S@C can be considered a form of homomorphic encryption, while the mask generation is performed in the TEE.

2.3 Specialized Hardware: GPUs, NPUs and FPGAs

Many computers contain specialized units that assist the general-purpose CPU. The specialized components can be on-board or extra components. In this thesis, we will use the term *fast processing unit* for anything that is faster than the CPU at a specific task. For example,GPUs are often employed in machine learning tasks.

We will discuss a couple of specialized units in this paragraph. The specialized units are usually co-processors next to a regular CPU. Of course there can be more than one co-processor in a PC, for example Intel's LunarLake PCs contain a CPU, a GPU and a Neural Processing Unit (NPU) [99].

2.3.1 Graphic Processing Units (GPUs)

Graphics Processing Units (GPUs) are used to render graphics and apply changes to them. GPUs were originally used to accelerate highly parallel, memory-intensive tasks like texture and polygon rendering, but acquired specialized units for geometric operations such as rotation and translation of vertices. Nowadays, GPUs include support for programmable *shaders*, which can manipulate vertices in a multitude of ways. They typically have a large amount of memory and a high number of processing cores, which allows them to perform massively parallel computations. As machine learning tasks involve large datasets and complex computations that do not change for each data set, parallel execution of tasks is very useful [176]. Graphic processing and machine learning share some characteristics: Many operations such as convolution and matrix operations are similar. Both types of computing work with few instructions on large amounts of data, following the *single instruction multiple data* (SIMD) principle. The resulting massively parallel computations can be significantly faster than CPU computations. Also, both graphical operations and neural networks work on floating point values, so no data type transformation is required. Many machine learning frameworks such as Pytorch and Tensorflow contain both CPU and GPU libraries out-of-the-box [2, 178]. The CUDA framework for NVIDIA GPUs provide several libraries for machine learning [1, 51].

2.3.2 Neural Processing Units (NPUs)

Neural Processing Unit (NPUs) are a recent type of specialized chip that came up with the rise of machine learning and artificial intelligence (AI). They are also called AI accelerators. NPUs are used in laptops, mobile devices, IoT devices and cloud computing servers [54]. They are used in large tensor nets in cloud servers as well as for computing-on-the-edge, where small devices perform on-board calculations to send processed data. As it is an emerging technology, no typical design has

been established yet. However, Intel shipped their first computers with an on-board NPU in October 2024, and Google has used NPUs for years by now [99, 148].

2.3.3 Field-programmable arrays (FPGAs)

Field-programmable Gate Arrays (FPGAs) are programmable hardware. They are a cost efficient way to try out new hardware designs and, as such, are used in evolving fields such as machine learning [11]. FPGAs are usually used for prototyping and not as long term solutions. As they are hardware, operations on FPGAs are usually very fast, almost as fast as on specialized chips. Programming FPGAs requires knowledge of hardware description languages (HDLs) such as VDL. There are open-source VDL libraries for machine learning [40]. While FPGAs offer fast performance and flexibility, they may also require significant expertise and resources to design and implement. FPGAs are relatively small and do not scale as well as specialized chips. They can however be reused for a different task.

2.4 Trusted Execution Environments (TEEs)

Trusted Execution Environments (TEEs) provide a secure environment on untrusted hardware by offering hardware-sealed, attested *enclaves*. The hardware isolates the enclave from other programs on the host without regarding privilege level, thus everything from a small subroutine to a whole VM can run securely, undisturbed even by a compromised operating system or hypervisor (we will get to hypervisors in Subsection 2.6.1). Common TEEs are Intel Software Guard Extensions (SGX) [49, 98, 145], AMD Secure Encrypted Virtualization (SEV) [111], and Sanctum [50]. The ARM Confidential Compute Architecture (CCA) [132] and Intel Trust Domain Extensions (TDX) [190] are still in development.

Besides strong memory isolation, a TEE offers remote attestation for both the data and the code deployed within a specific enclave. Multiple mutually distrustful parties can compute on untrusted hardware while using sensitive data, making TEEs attractive for use in cloud computing service such as MLaaS [44, 89]. While there has been some discussion about the future relevance of TEEs after Intel discontinued SGX on consumer devices, the efforts of both Intel and other vendors

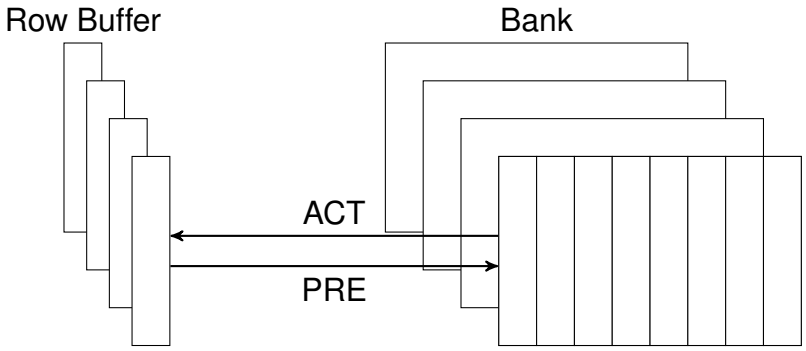

Figure 2.2: DRAM organization in banks and rows. A PRE (pre-charge) command activates or closes a bank, an ACT (activate) command loads the row into the row buffer. Figure from [88].

to build alternatives is evidence that TEEs will be an important feature in the future.

We worked with Intel SGX instead of newer TEEs such as TDX or AMD SEV to minimize the trusted computing base. SGX applications need to be divided into trusted components which run within the enclave and untrusted components that run outside of it. The developer defines the interface between the two. The trusted part should be kept small to minimize the potential attack surface. Keeping the context switches between the trusted and untrusted parts to a minimum increases performance.

2.5 DRAM Organization

The main memory is *Dynamic Random Access Memory (DRAM)*, an array of memory cells which consist of one transistor and one capacitor per cell [208]. A charged capacitor represents a 1-bit, while a discharged one is a 0-bit. As capacitors lose charge due to leakage, they need to be refreshed constantly every few milliseconds (currently roughly every 8ms) to keep their charge. The process called *row refresh* is performed one row at a time, just by reading the row [88]. As only one transistor and one capacitor are required for each bit, DRAM modules have a high density. So on the upside the chips can be very large, but on the downside they are comparably slow and, as mentioned above, require constant refreshing.

The DRAM organization is illustrated in Figure 2.2. We can see multiple *memory banks*, which are then subdivided into the aforementioned rows. Depending on the hardware configuration, a number of physical address bits are used to map memory pages to banks [179]. Rows are generally placed sequentially within the banks, which in turn means that access to continuous physical memory will grant access to adjacent rows. Accessing a memory location causes the corresponding row to be activated (and thus refreshed) and loaded into the row buffer. Subsequent requests for the same row will then be served from the buffer.

Rowhammer attacks amplify the capacitor leakage for malicious purposes. Let's assume a target row A. Kim et al. showed that accessing neighboring rows repeatedly will enhance the leaking of the charge of row A if it is faster than the row refresh [119]. If the data is not refreshed before the charge is lost, bit flips can occur in row A. The corruption of data in a row not controlled by the attacker is an attack on the integrity of the data or the availability of a service [76]. By placing security-critical data or code within row A, Rowhammer attacks can additionally be used to enhance privileges [196, 233].

Another security risk for DRAM is direct data reconstruction via *cold boot attacks*. They are used to recover data from DRAM once it is without power and even removed from the computer. While it is often assumed that data recovery from DRAM is not possible, Link and May actually discovered that cooling the DRAM instantly after removing the power supply allowed them to retain data for up to a week afterwards as early as 1979 [134]. In 2009, Halderman et al. showed the complete reconstruction of an AES key in a cold boot attack [83].

2.6 Process Isolation

A central paradigm of computer security is *process isolation*. It provides a higher level of security as well as performance by issuing resources to one process explicitly. If one or more processes require the same resource, they need to be *scheduled*. Processes should not have insight into each other's data, so memory needs to be split between them. Different address spaces (see Subsection 2.6.2) ensure this. The memory of a single process is further split between *kernel space* and *user space*. Splitting the memory is necessary to ensure that a process cannot elevate it's own privileges. If there is only one operating system on a machine, the

scheduler and the *memory management unit* fulfill these tasks. *Virtual machines* (further explained in Subsection 2.6.1) offer an additional level of isolation. They are managed by a *hypervisor*, and offer the additional benefit of running their own operating system. That means that different operating systems can run on the same physical machine.

2.6.1 The Hypervisor

On a cloud machine, processes run in Virtual Machines (VMs). The software responsible for creating and managing these VMs is called the *hypervisor*. It offers an additional layer of abstraction between the physical hardware and the VMs, allowing multiple VMs to run on the same physical machine. The hypervisor is responsible for the allocation and distribution of resources as well as the isolation between the virtual machines. As it requires access to memory management, process planning, I/O stack, network components and so on, the hypervisor requires elevated privileges on kernel level. In-kernel hypervisors are common, but pose a security risk: If the hypervisor is compromised, it can corrupt it's VMs, bypass data protections and give the adversary control over processing [207]. A compromised hypervisor poses a danger to the security goals confidentiality, availability and integrity [39, 216].

In 2023, Chen et al. showed a CVE rate of 7.9 per year for KVM and 24.7 per year for Xen. In 2025, this rate has risen to 19.4 and 32.4 CVEs per year respectively. For comparison, OpenSSL has CVE rate of 20.6 per year[2]. These numbers prove that hypervisors are an attractive target [39].

Hypervisor security should be ensured through basic security measures such as access control and isolation of networks. More inspired ideas to ensure hypervisor security go from detecting hypervisor introspection to adding more privilege levels and splitting the applications between them [39, 216].

[2]We are aware that CVE per year is not an objective metric as it depends on what the attacked component considers part of their threat model, as well as people following responsible disclosure procedures, but it provides an estimate of the attack surface and the specific component's attractiveness for attackers.

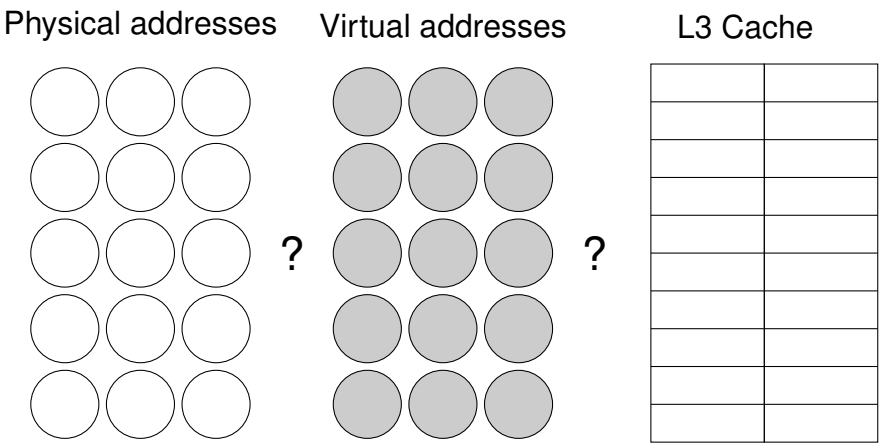

Figure 2.3: The mapping between physical addresses, virtual addresses and the L3 cache is unknown

2.6.2 Memory Management and Virtualization

Processes work with *virtual memory*. Virtual memory is more versatile than physical memory, as the computer can issue more memory than it actually has. Different processes can each get their own share of virtual memory, which may be in very different places physically. Virtual memory is always presented to the process as a continuous address space, regardless of the fragmentation state of the physical memory. In addition to veiling the system state, developing with continuous memory is easier for users.

Virtual memory is managed by the *Memory Management Unit (MMU)*. It assigns isolated virtual address spaces to all running tasks, thus splitting the physical DRAM. Memory is typically allocated in 4 kB *pages* The mapping between the virtual page and the DRAM page is held in the *page table*. The least significant 12 bits of the address are called the *page offset* and are not translated, meaning they are the same in the virtual and the physical address. When an address gets translated, the result is stored in the *Translation Look-aside Buffer (TLB)*. Repeatedly accessing the same address will thus be faster, as translation is not necessary any more.

The mapping between physical and virtual addresses is *hidden* from unprivileged processes. They can only access their own memory space, not the kernel address space or the address space of other processes.

As illustrated in Figure 2.3, the physical addresses are used for mapping memory to cache sets (see Section 2.9, the process also cannot determine their data in the lower level caches, which needs to be done by the MMU. Libraries that are cached once for several processes will thus have different virtual addresses coherent with the other process addresses, not spoiling information about co-located processes. However, the libraries are only present once in physical memory. An attacker will also find it harder to determine which memory addresses are physically close, which is for example required for Rowhammer-style attacks [78]. Within each processes virtual address space, the *kernel* has an extra, privileged area. The kernel cannot be directly accessed by the process, preventing the attacker to intercept I/O signals from the user [136, 162, 173, 186]. So, beside the ability to offer more memory than actually present and offering fragmented memory as if it was continuous, memory virtualization also adds a layer of security [16]. As we will see in Section 2.10, this can be circumvented if out-of-order execution occurs.

2.7 Memory Order Buffer (MOB)

Since DRAM storage is slow compared to the processing speed of the CPUs, memory operations need to be organized efficiently in order to avoid a memory bottleneck. The *Memory Order Buffer (MOB)* is responsible for this task. It is tightly coupled with the data cache, which we will inspect in the next section. On Intel processors, the MOB applies the Intel memory ordering rules [6, 7, 146]. It follows two principles:

1. Memory `stores` are executed in-order.
2. Memory `loads` can be executed out-of-order.

The first rule ensures correct memory accesses on commitment. The second improves efficiency by loading data that is already available while waiting for data that needs to be fetched from DRAM. The MOB includes two buffers. Firstly, the *store buffer* consisting of the *Store Address Buffer (SAB)* and the *Store Data Buffer (SDB)*. We use the term *Store Buffer* to mention the logically combined SAB and SDB units. The second buffer is the *load buffer*. The store buffer enables the processor to continue working while finished operations are still being committed to memory. The pipeline thus does not have to wait for the `store` to complete.

The MOB can also use the store buffer to execute `load` instructions out-of-order: Data that is not yet committed to memory can be loaded from the store buffer to continue the calculation in a mechanism called *store forwarding*. Intel's design of the store buffer is not documented, but there is evidence that suggests that it only holds the virtual address as well as part of the physical address [6, 7, 124]. As the MOB does not contain the full address of the `store`, forwarding the `load` is a *speculative* process. It may be based on a false dependency and thus require rolling back. We will get back to the dependency resolution in Chapter 4.

2.8 Transactional Memory and Intel TSX

Transactional memory is a countermeasure to data conflicts. The idea of transaction is rooted in database engineering: When trying to update a number of tables at once, either all updates go through or none of them. Similarly, data in a read set is locked for writing during the transaction, and vice versa. The concept was adapted to web services, where more concurrent users and a potential time delay make transactional behavior even more necessary. For example, while a customer is in the middle of booking a hotel room, that particular entry is blocked for every other user. Blocking certain data exclusively for one process works directly orthogonal to performance enhancing mechanisms such as speculative execution or store-to-load-forwarding. The programmer specifies code regions to block from other processes. The processor then tries to execute the code. If it can complete the transactional regions without data conflicts it will and commit the result. If not, the entire process is rolled back. The concept is called *optimistic execution* of code regions: The programmer can be optimistic that no conflicts with other threads or CPUs cores will occur. Note that the data is not actually locked, as all other processes can still access it. Instead, only the transactional code itself is slowed down by a callback.

Transactional memory uses a hardware-based callback mechanism in case of a conflict. It works on local copies of data and, if the execution finishes without a conflict, all memory changes are committed atomically and thus become visible to other processes at once. Otherwise, if a conflict occurs, the transaction is canceled and all memory changes are discarded and a callback function is triggered. The process is called an *abort* and the callback function is the *abort handler*.

Disselkoen et al. list the main reasons to abort a transaction in Intel TSX from the Intel developer guide in 2016 and their own findings, combined with undocumented behavior described by Dice et al. [56, 57]:

1. Executing certain instructions, such as CPUID or the explicit XABORT instruction
2. Executing system calls
3. OS interrupts
4. Nesting transactions too deeply
5. Access violations and page faults
6. Read-Write or Write-Write memory conflicts with other threads or processes (including other cores) at the cache line granularity—whether those other processes are using TSX or not
7. A cache line which has been written during the transaction (i.e., a cache line in the transaction's "write set") is evicted from the L1 cache
8. A cache line which has been read during the transaction (i.e. a cache line in the transaction's "read set") is evicted from the L3 cache

Let us take note of a couple of points: Firstly, because of reason 3, transactions may abort without any conflict of interest at all. Since the OS frequently interrupts processes, this abort reason also limits the length of a single transactional region to the amount of instructions that can be performed between two OS interrupts. Secondly, reason 7 and 8 concern the cache, which will become important in Subsection 2.9.1. Depending on the programming, the transaction can then be repeated or another action may occur. While transactional memory ensures conflict-free programming, it may slow down the protected code significantly.

While the Intel implementation of transactional memory, Intel TSX, has been used to conduct timer-free cache attacks [57], it has also been leveraged to protect processes against cache attacks [75] or to prevent data input modification and thus protect against the exploitation of double fetch bugs [193]. Li et al. use TSX transactions to protect private RSA keys from both software and hardware attacks such as coldboot attacks. They do however also mention that protecting code with TSX opens the door for DOS attacks, as constantly interrupting the protected code means it will be unlikely to finish execution successfully in a timely manner [131].

The same problem is not discussed, but also an issue in the work by Gruss et al. [75].

After several attacks, Intel issued a microcode update disabling TSX on all machines in 2021 [101]. It can however still be turned on with root privileges. As discussed at an Intel conference in 2021, the company does continue to develop it and it is likely it will be re-enabled in future versions.

2.9 Caches and How They Work

As mentioned in Section 2.5, a DRAM storage is comparatively slow. To achieve better performance, small memory blocks called *caches* are introduced between the processor and the main memory. They are typically *static random access memory (SRAM)* components, which contain circuits instead of capacitors. Caches are thus very fast and keep their content for a long time without refreshing [208]. There are several caches on most modern processors, organized hierarchically in different *levels*. The lower level (L1 and L2) caches are core private and closest to the processor, reducing latency due to data transfer. The last level cache (LLC or L3) is shared among cores. In this work we only discuss inclusive caches: all data in the private low-level caches also has to be in the shared LLC.

As the caches are by nature smaller than the main memory, several challenges need to be addressed when building and analyzing them:

- The *cache mapping* determines which data is stored in which part of the cache. It is described by the *associativity*, *wayness* and *number of sets* of a cache.
- The *replacement policy* is closely connected to the cache mapping. It is an algorithm to figure out which data gets removed from the cache if the cache is full and new data is requested.
- To further increase performance, *prefetching*-strategies can be implemented.
- The *cache coherence* ensures that the data in the cache is up to date and stored correctly when needed.

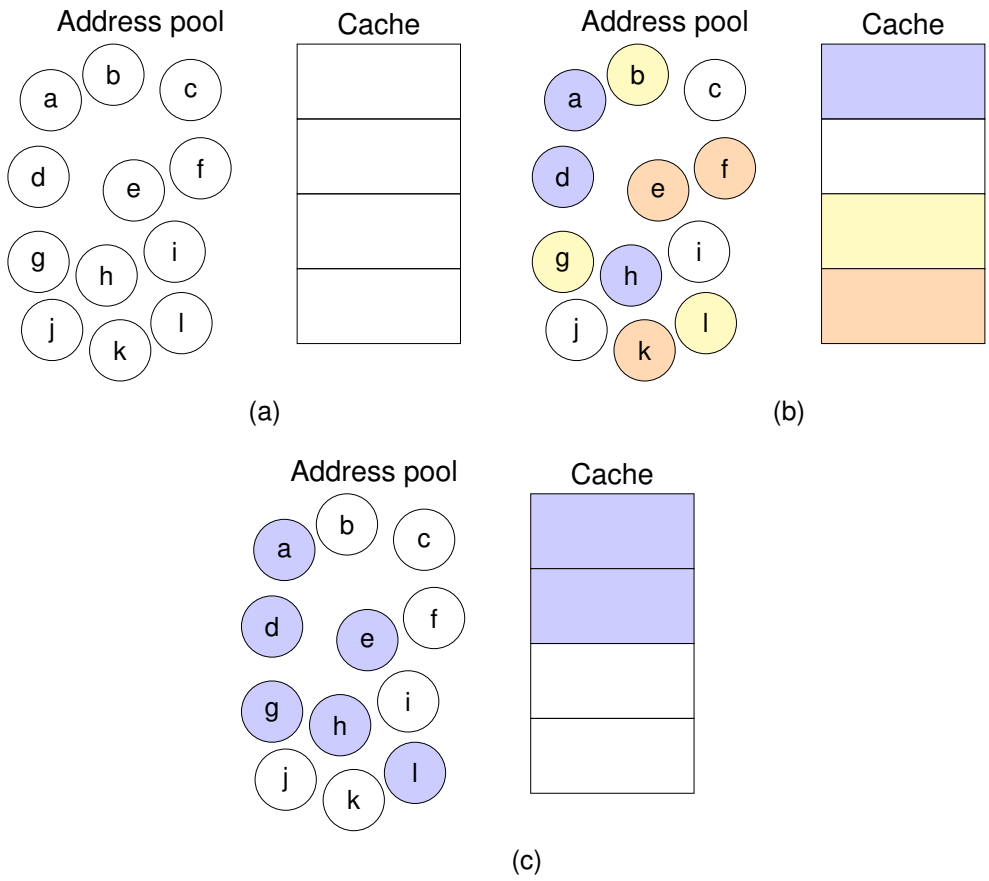

Figure 2.4: The cache mapping defines which memory address can be placed in which region of the cache. (a) Fully associative: Every address can go everywhere (b) Directly mapped: Every address can go to exactly one place (c) Set associative: A set of addresses can go into a specific cache location.

We will focus on the first two points. Caches are organized in *lines*, the smallest unit that is fetched from main memory. They are usually 64 bytes. Different cache mappings are depicted in Figure 2.4. The easiest cache mapping is a *fully associative* cache, where each memory block can be placed at any cache line. However, in a large cache, this policy leads to high performance cost to determine whether an element is in the cache. The other extreme is a *directly mapped* cache, where each cache line can be placed at exactly one location in the cache. A direct mapping can lead to very high cache miss rates. The middle ground is grouping lines together in a *cache set* s. The size w of a cache set is also called the *wayness* of the cache. Many caches additionally group the sets into *slices*. The mapping of a cache line

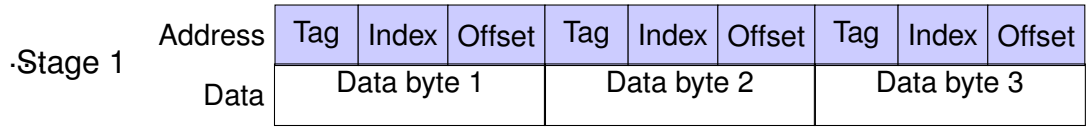

Figure 2.5: A single cache line. The address is split into tag, index and offset. The offset and the index determine the cache set the line is placed in.

to its cache set is based on the physical address of the cache line. The address bits are divided into *offset*, *index* and *tag*. The offset are usually the lowest-order 6 bits used to locate data within a 64 byte line. The offset-bits are the same in the physical and virtual address. The index bits are $\log_2(s)$ consecutive bits starting from the offset bits. The index bits determine the cache set the line is placed in. The tag are the remaining bits of the address. The tag identifies whether the data is cached. An example is shown in Figure 2.5.

Cache mappings are often not publicly documented, and reverse engineering them is an active research area [144]. A fully associative cache can also be seen as a cache with a single set, while a directly mapped cache can be seen as a cache with wayness 1.

When a process requires data, the MMU and the TLB translate the virtual address to a physical address. The cache controller determines whether this particular cache line is in the cache. If it is, a *cache hit* occurs. If the data is not in the cache, a *cache miss* occurs, and the data needs to be fetched from main memory. The data is then placed in the cache, requiring an existing cache entry to be evicted in the process. The *replacement policy* determines which cache entry is evicted, thus placing the new memory block in the cache. The replacement policy is closely connected to future cache hits and misses and hence responsible for achieving high performance. Replacement policies are often not published, and considerable efforts have gone into reverse engineering them [3, 4, 25, 231]. Understanding the replacement policy is crucial for many cache or fault injection attacks. In this thesis, the criticality of understanding the cache replacement policy to launch a successful, precise cache attack will become clear in Chapter 5.

Replacement policies work on two assumptions: *spatial locality* and *temporal locality* [88]. Spatial locality assumes that data located next to each other in memory will likely be used together. If a data structure is larger than a single cache line, spatial locality is automatically true. Spatial locality is leveraged in the set associativity, as

memory blocks with adjacent physical addresses will be mapped to different cache sets and thus do not evict each other. It is also used by prefetching mechanisms, which load data into the cache before it is actually requested. Temporal locality assumes that data that has recently been used will likely be used again soon. It directly leads to one of the most common replacement policies, the *Least Recently Used (LRU)* policy. Other policies count the data use over a certain time and keep the most *frequently* used data, or evict that data that is oldest without considering the time or frequency of use (first-in-first-out) [184]. There are mixed forms including random elements as well as Pseudo-forms like Most Recently Used (MRU), which is an LRU policy with a toggle instead of a counter also called *pseudo-LRU* policy. We will discuss a tree-based pseudo-LRU policy in Chapter 5. Overall, we will only consider LRU policies in this work, as they are implemented in all our test machines.

2.9.1 Cache Side Channel Attacks

The last level cache is shared between processes and can thus be leveraged for microarchitectural side channel attacks, either by directly inferring information or by using the cache as a covert channel to communicate information between processes. Ever since the cache was first mentioned as a side channel by Hu in 1992, many cache side channel attacks have been published [25, 68, 79, 92, 139, 156, 236].

Flush+Reload is an early cache side channel attack developed by Gullasch et al. and Yarom and Falkner [81, 236]. It requires shared memory between the processes, which can be a shared library or something similar. The attacker *flushes* cache lines from the cache using an instruction such as `clflush` in Intel processors. She then waits for the victim process to execute, then reloads the block and measures the time before it is available. If the reload time indicates a cache hit, the attacker knows the victim used the flushed memory block. If it indicates a miss, the victim did not use the memory block. The *Flush+Flush* technique uses a similar approach, but measures the execution time of the flush instruction instead [79].

The *Flush+Reload* attack has a high resolution as it can detect usage of individual cache lines. It is easy to implement and interpret. However, it uses the flush instruction, which is not present in some processors and not available on all levels of the software stack. For example, it is not available in browsers [68]. It also

requires shared memory, which is not always the case for the victim and attacker processes.

In the absence of a flush instruction, a *Prime+Probe* attack can be used. It can convey data with cache-set granularity. The attacker *primes* a cache set by completely filling it, then waits for the victim process to execute. Then the attacker *probes* the cache set by reloading all the addresses in the cache set again. If the access time indicates one or more cache misses, the victim accessed the cache set in question [95, 139]. *Prime+Probe* does not require any specific OS properties, so it can be applied to virtually any system. In preparation for *Prime+Probe* the attacker needs to determine a minimal set of addresses that fill a specific cache set, a so-called *eviction set*. An eviction set contains w different addresses in a w-way set-associative cache. As the address information for cache lines is not available for a user-level process, working out missing address information as well as set- and slice-selection is a large research area [96, 103, 139, 221]. In this thesis, eviction set creation and address resolution will be discussed in depth in Chapter 4.

As measuring the time difference between a cache hit and a cache miss is a crucial part of both *Flush+Reload* and *Prime+Probe* attacks, fairly precises timers are required to execute these cache attacks. As a consequence, disabling access to timers was considered a valid countermeasure for cache attacks for a while. Disselkoen, Kohlbrenner, Porter, and Tullsen constructed a timer-less cache attack in 2017 by leveraging Intel Transactional Memory (Intel TSX). In Section 2.8 we discussed the abort reasons for a TSX transaction. There are several cache-related abort reasons, and the *Prime+Abort* attack uses the fact that a transaction aborts if data from its *write set* is evicted from the L1 cache or if data from its *read set* is evicted from the L3 cache [56, 57]. It follows that an abort signal of the attacker process indicates a cache set access of the victim process. The *Prime+Abort* attack achieves the same resolution as *Flush+Reload* but obviously requires Intel TSX on the victim machine.

Side channels can be used constructively as well: Tuzel et al. use cache side channels to detect hypervisor introspection in cloud services [216]. Briongos et al. use a cache covert channel to protect enclaves against forking attacks [24].

2.10 Speculative Behavior

When talking about caches, we also discussed replacement policies. A good replacement policy tries to make sure the data required by a program in the near future is already available. It thus basically guesses the next required data based on the previous data usage. Trying to predict future situations is a concept that is applied widely in computer architecture. We will discuss a few examples in the following.

Speculative execution tries to predict the control flow of a program. We will look in detail at an example misusing the *dynamic branch predictor*. The branch predictor is a piece of hardware that stores which branch was taken in a particular region of code. It then predicts which branch will be taken next. Branch prediction is a fairly intuitive concept for programmers: In a *for* loop with limit 100, we take the branch into the loop 100 times, and the branch out of the loop only once. After a few iterations of the loop, it seems like we "always" take the branch into the loop. The processor can thus execute the data within the loop before actually checking if the limit is reached, and will still be correct 99% of the time. The processor executes a part of the code *speculatively*. If it later turns out that the limit was reached, the speculatively executed code is rolled back and the program continues with the correct branch. As Kocher et al. and Lipp et al. found out with their famous attacks SPECTRE and MELTDOWN in 2018, only the architectural state is rolled back, but the microarchitectural state such as caches remains the same [121, 136]. Observing the microarchitectural state allows an attacker to leverage speculative behavior to infer secret information.

Let us imagine a program containing a conditional branch. A toy example is shown in Figure 2.6: If the parameter *limit* is below the size of a dummy array, we access the dummy array at that point. Note that contrary to many attacks, the victim function is not a vulnerable library function that can be repaired, but a function in the attack code.

To leverage this function in an attack, we assume that the dummy array is placed in memory just before some secret data. It is also possible to scan more distant parts of the memory. The attack function is shown in Figure 2.7. First, the branch predictor is trained: we call the *victim_function* with an allowed value for *limit*, so it takes the path into the conditional branch several times. Experiments show that four to five training iterations are enough on most hardware to program the branch

```
1: function VICTIM_FUNCTION(LIMIT)
2:     if (limit < dummy_array_size) then
3:         read_array[dummy_array[limit] * 256] = 1
4:     end if
5: end function
```

Figure 2.6: The conditional branch of this function is vulnerable to attacks using speculative execution. The array *dummy_array* is an array with meaningless content that inhabits a memory location in front of secret content. The array *read_array* is used to transport information out of the victim function.

predictor. It will then always predict that the control flow will take the path into the conditional branch. At that point, we increase the counter once more. It is now larger than limit, so larger than the size of the dummy array.

```
1: function TRAIN_AND_SPECULATE(LIMIT)
2:     counter = 0
3:     while (counter < limit+1) do
4:         victim_function(counter)
5:         counter++
6:     end while
7: end function
```

Figure 2.7: Example algorithm for training a branch predictor to speculate past a limitation within the victim function.

Looking back to victim function in Figure 2.6, we can see that this means a memory location *outside* of the allowed limit of the dummy array is accessed. As the branch predictor is trained to allow that, the code will *speculate* past the condition *before* finishing the evaluation of the condition. We can thus access a memory location outside of our program's memory space. By setting a marker in our read array, we can infer the value of the byte we accessed. While the read array will be rolled back, the cache will still hold the entry of the read array. We can thus infer the access via a cache attack, for example *Flush+Reload*.

There are many other examples of speculative behavior. For example, processors speculate about exceptions occurring or return points after jumps [121]. The store forwarding discussed in Section 2.7 is also a speculative process: It forwards data before it fully resolves the address, thus requiring a timely rollback process that opens a new side channel.

3

CARNIVAL: Generating Secret Masks from Public Knowledge

In the introduction, Alice was filling out her loan application. It was stated that the bank might use a machine learning model to calculate her credit score. While we only used this example to illustrate our motivation, many banks actually do use machine learning and related techniques for various tasks, including decision making [8, 202]. It means that private data is processed on a server out of the *data owner's* control. Protecting this data and the output during the machine learning inference process is the goal of CARNIVAL, a framework developed in a joint work with Sebastian Berndt, Jonas Sander, and Thomas Eisenbarth. CARNIVAL improves the offline-phase of the neural network inference framework SLALOM by Tramer and Boneh [213]. SLALOM offers private inference by applying masks to the inputs and outputs of a neural net, as well as integrity guarantees for the calculation. It uses a Trusted Execution Environment (TEE) and a Graphics Processing Unit (GPU).

This chapter is mainly based on work published in [29] and some parts of [189]. Parts of this chapter are taken from these publications. Within those works, I developed the CARNIVAL primitive, which generates secret masks by combining many publicly known masks. Sebastian Berndt and I worked on the security discussion while Jonas Sander implemented benchmarks to evaluate the performance. In addition to the theoretical, cryptographically proven security, we also ran experiments regarding the uniformity of masks generated with smaller keys. The experiments for this part were conducted by Frederik Lehmann as part of his Bachelor thesis. In [189], we developed DASH, a second framework building on SLALOM. For DASH, Jonas Sander developed GPU-friendly garbled circuits, which are applied to provide the same security guarantees as in CARNIVAL without using masks. I worked on the security scenario as well as the related work and editorial tasks.

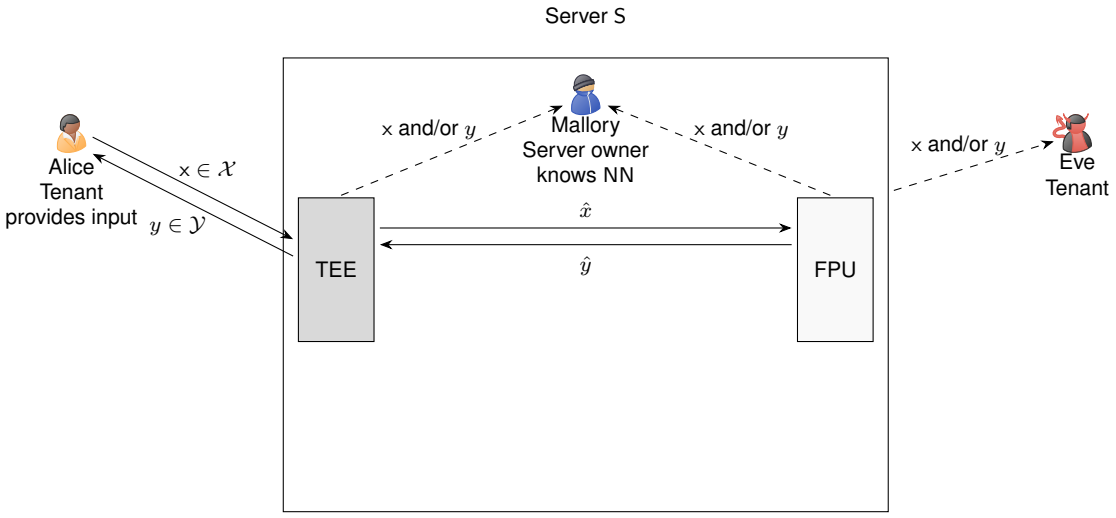

Figure 3.1: The setup for CARNIVAL: Mallory host a server S on which a pretrained neural net NN waits for input to infer on. Alice is an honest tenant and sends input x to receive output y. Both Eve, an attacking tenant, and Mallory try to extract information about x and y. As the CPU and LLC are not relevant for our scenario, they are omitted from the setup.

As discussed in Chapter 1, machine learning is a vastly growing field. ML has several particularities that make it a prime candidate for outsourcing to shared hardware: First of all, the training of a network requires extensive amounts of data and computational resources. The data can be confidential. After training, the computational requirements are reduced drastically, but the readily trained model is a new asset that may need protection.

Besides being a business asset, a trained network is often large, which makes *private inference* a substantial research field. We focus on *hardware-assisted secure outsourced inference* for neural networks. While the term Deep Learning would be technically more specific, related work instead uses the umbrella term machine learning [36, 38, 55, 59, 110, 213, 226]. The term inference is also used in literature instead of the more accurate term evaluation. To embed this work into the context and semantics typically used in this research area, this thesis continues applying the terms used in related work.

We need to adapt our basic attacker model from Chapter 1 to visualize the scenario in CARNIVAL. The adapted version is depicted in Figure 3.1. Alice provides the input and receives the output. Both the server owner Mallory with elevated privileges and

Eve, another tenant, want to extract the input and output. The CPUs and caches are not relevant for CARNIVAL and are thus omitted.

3.1 Assumptions and Theoretical Foundation

Let us assume a model owner, for example Alice's bank, has spent resources to train and refine a model. The neural network $\mathsf{NN}\colon \mathcal{X} \to \mathcal{Y}$ consists of layers $\mathsf{NN}_i\colon \mathcal{X}_i \to \mathcal{Y}_i$ and each layer consists of a linear transformation Lin_i followed by a non-linear activation function NLin_i. The linear layers are usually matrix multiplications or convolutions. As convolutions can be converted to a matrix multiplication, we will use the matrix multiplication as the exemplary function in this work. Furthermore, we assume that the input $\mathcal{X}_i$ and the output $\mathcal{Y}_i$ are represented by Integer vectors. The model owner uses a server S to host NN and now offers inference as a service to clients. Using IaaS has two reasons:

- The model owner does not want to share NN, which is a business asset. We note here that strong model extraction attacks exist (e.g., [34, 177, 214]) and thus do not focus on model privacy in this work.
- The client C on the other hand does not have the computational resources to infer on the model, and might even lack the knowledge to deploy it.

The client C provides the input data $\mathsf{x} \in \mathcal{X}$, which is sensitive information not to be shared with the server owner S or other tenants. The goal of the client is to compute $\mathsf{NN}(\mathsf{x}) \in \mathcal{Y}$ with minimal computation costs by making use of S without revealing information about the input x or the output $y = \mathsf{NN}(\mathsf{x})$ to the server owner or model owner. We thus protect the security goal of confidentiality by providing *input privacy* and *output privacy* respectively.

To achieve privacy for the client, the model owner offers a Trusted Execution Environment (TEE) within the server. The client can load the data into the TEE, which is then protected from a potential malicious model owner. As usual, the code within the TEE can be checked by the client via remote attestation and the data processed within the TEE is constantly encrypted in memory. Inferring purely on the TEE would however create massive performance issues, either increasing the MLaaS prices or rendering the model owners business model useless. One approach to handle this problem is by adding a *Fast Processing Unit (FPU)* to the server. An

FPU is anything faster than the regular processor, so it can be a GPU or specialized hardware such as an FPGA.

Having processing units with different computational performance creates the challenge for the model owner to move as much computational load as possible to the FPU *without* compromising the client's data. In order to protect the client's data, some operations still need to be computed in the TEE, which is slower than the FPU by orders of magnitude. Hence, the main question of this line of research is to minimize the amount of computations required in the TEE while still guaranteeing privacy to the client.

One solution to outsource the computational load to the FPU was presented by Tramèr and Boneh [213], who developed a framework called SLALOM that allows a private computation of $\mathsf{NN}(\mathrm{x})$ if the server is equipped with a secure TEE. To do so, they use the *offline-online* model inspired by very efficient MPC protocols such as BDOZ [18] or SPDZ [53]. In this model, the computation is split into an offline or *preprocessing* phase that is independent of the sensitive input x, where the TEE generates some masking values and their counterparts (later referred to as unmasking values) for the client. These values, combined with x are then used in the online phase to produce the output $\mathsf{NN}(\mathrm{x})$. The preprocessing phase is implemented solely on the TEE and not included in the performance evaluation.

In this work, we design a method to perform the preprocessing phase more efficiently using the Subset sum problem over finite fields as a randomness generator.

3.1.1 The Subset Sum Problem

The Subset sum problem is a well-known NP-hard problem. Given a finite field F_p containing the numbers $0, 1, 2, \cdots p-1, p$ and a set $S = \{s_1, \ldots, s_n\} \subseteq F_p^n$ of field elements and a function

$$f(k) = \sum_i k_i s_i \mod p$$

that maps a binary vector k to another field element, the goal is to reconstruct the binary characteristic vector k. From a cryptographic perspective, $f(k)$ is strongly believed to be a *one-way function* (for appropriate choice of the parameters n and p), meaning that there is no efficient algorithm to recover k from $f(k)$. Furthermore,

f can be used as pseudo-randomness generator under certain conditions [21, 94, 150], which CARNIVAL leverages to generate pseudo-random One-time Pads (OTPs) as masks. We will provide a detailed description of the process in Section 3.3. The high-level improvement is the swapping of matrix multiplication in the TEE for matrix additions only, which reduces the computational complexity and runtime drastically. As an added bonus, the FPU is utilized during the preprocessing phase as well, leading to better resource balancing.

3.1.2 Knapsack Functions in Cryptography

The Subset sum problem is a specific variant of the knapsack problem, namely a 0-1 knapsack problem where the weights of the knapsack items equal their profit [181]. Knapsack functions have a long history as cryptographic functions. Many cryptographic systems based on this function family have been introduced [45, 113, 130, 147]. While knapsack functions lead to one-way functions with very useful properties [149], using them in asymmetric scenarios has been a very difficult task, as shown by many broken systems [171, 172, 197]. One of their most useful properties is the fact that knapsack functions can be used as pseudo-random number generators or hash functions [94], with a number of properties that make them interesting candidates for various practical scenarios:

- Knapsack functions are able to generate pseudo-randomness relatively easy, using additions only.
- The highly parallel instructions over a finite field can be implemented very efficiently and
- Linear operations on the knapsack elements result in linear transformations of the resulting one-time pad, making them suitable for homomorphic operations.

Most of the asymmetric knapsack cryptosystems devised so far use a specially constructed set S as the public key and additional information about S as private key. In these systems k is the plaintext and $f(k)$ is the ciphertext [130, 147]. As usual in public key crypto systems, the ciphertext can only be deciphered efficiently with the private key, namely the additional information about S. Hence, the trapdoor information is concealed in S [126].

In contrast, we do not use S to conceal any information, but use k as the key to generate an OTP which is then used to encrypt the message, generating a symmetric cryptosystem or masking scheme.

3.1.3 Formal Definitions and Notations

We will consider outsourcing the computation of $g(x)$ for a sensitive value x and a publicly known function g to an untrusted party. We follow and extend the definitions by Tramer and Boneh [213]. A *secure outsourcing scheme* for a function $g\colon \mathcal{X} \to \mathcal{Y}$ between a client C and a server S consists of three algorithms: $\mathtt{Setup}$, $\mathtt{Preproc}$, and $\mathtt{Online}$. The algorithm $\mathtt{Setup}$ is called a single time and produces some public parameters $\mathtt{param}$. Given these parameters, $\mathtt{Preproc}$ can be used by C to produce some data-independent state $\mathtt{state}$. Finally, $\mathtt{Online}$ is an interactive protocol between C providing the inputs $\mathtt{param}$, $\mathtt{state}$, and the sensitive information $\mathsf{x} \in \mathcal{X}$ such that at the end of the protocol, C receives a value $y \in \mathcal{Y}$ or aborts the protocol. We denote the result of $\mathtt{Online}$ obtained by C when using inputs $\mathtt{param}$, $\mathtt{state}$, and x and running on S by $\mathtt{Online}_{\mathsf{C},\mathsf{S}}(\mathtt{param}, \mathtt{state}, \mathsf{x})$. Such a scheme needs to fulfill several important properties such as:

Correctness: For any $\mathtt{param}$ produced by algorithm $\mathtt{Setup}$, any $\mathtt{state}$ produced by $\mathtt{Preproc}(\mathtt{param})$, any $\mathsf{x} \in \mathcal{X}$, and any $y = \mathtt{Online}_{\mathsf{C},\mathsf{S}}(\mathtt{param}, \mathtt{state}, \mathsf{x})$, we have $y = g(\mathsf{x})$. Similar to SLALOM, to prevent a malicious server from using a different function $g' \neq g$, we could make use of zero-knowledge proofs.

Privacy: For any $\mathtt{param}$ produced by algorithm $\mathtt{Setup}$, any $\mathtt{state}$ produced by $\mathtt{Preproc}(\mathtt{param})$, any $\mathsf{x} \in \mathcal{X}$, any $\mathsf{x}' \in \mathcal{X}$, and any probabilistic polynomial-time algorithm running on S, the views of S in $\mathtt{Online}_{\mathsf{C},\mathsf{S}}(\mathtt{param}, \mathtt{state}, \mathsf{x})$ and in $\mathtt{Online}_{\mathsf{C},\mathsf{S}}(\mathtt{param}, \mathtt{state}, \mathsf{x}')$ are computationally indistinguishable.

t-Integrity: For any $\mathtt{param}$ produced by algorithm $\mathtt{Setup}$, any $\mathtt{state}$ produced by $\mathtt{Preproc}(\mathtt{param})$, any $\mathsf{x} \in \mathcal{X}$, and any probabilistic polynomial-time algorithm running on S, the probability that $\mathtt{Online}_{\mathsf{C},\mathsf{S}}(\mathtt{param}, \mathtt{state}, \mathsf{x}')$ does lead to a value $y' \in \mathcal{Y}$ with $y' \neq g(\mathsf{x})$ is at most t.

For clarity and ease of understanding, we provide a summary of the key variables and notations used throughout this chapter.

S	The server from our attack scenario. It is controlled by an honest-but-curios party and contains a TEE as well as an FPU.
C	The client from our attack scenario.
$\mathsf{x} \in \mathcal{X}$	Unchanged client input from the possible input range $\mathcal{X}$. For the Neural Network NN in our scenario, this is a vector.
x_i	Input for layer i. x corresponds to x_0.
y_i	Output of layer i. $y_i \in \mathcal{Y}$ for a network of n layers corresponds to the output the client receives.
$\hat{x}$, $\hat{y}$	A masked (possibly intermediate) input x or output y.
S	Set of elements s_i from finite field $\in F_p^n$ that can be added up in the Subset sum problem. See Subsection 3.1.1 for details.
$S_m[j]$	Set of possible masks for layer j. The masks are random values m_i. One S_m is required for each linear layer of the network.
$S_u[j]$	Set of possible unmasking values for layer j. The u_i are calculated on the FPU by inferring a layer on m_i. One S_u is required for each S_m, thus for each linear layer.
W	Weight matrix of a linear layer in NN.
$k[j] \in 0, 1^n$	Key to create the OTP for layer [j] by adding up the $m_i \in S_m[j]$ where $k_i = 1$.
$\sigma[j] \in$ NLin	Non-linear layer following the linear layer j in the neural net.

3.2 The Baseline: SLALOM in Detail

SLALOM is a hybrid scheme for outsourced machine learning inference that elegantly combines homomorphic encryption using one-time pads for the linear layers and a TEE for the non-linear layers. It guarantees the integrity of the inference and has an optional privacy feature for input and output privacy. We only consider the privacy scenario. SLALOM works in the setting described in Figure 3.1. The general approach of SLALOM is illustrated in Figure 3.2 and can be summed up as follows:

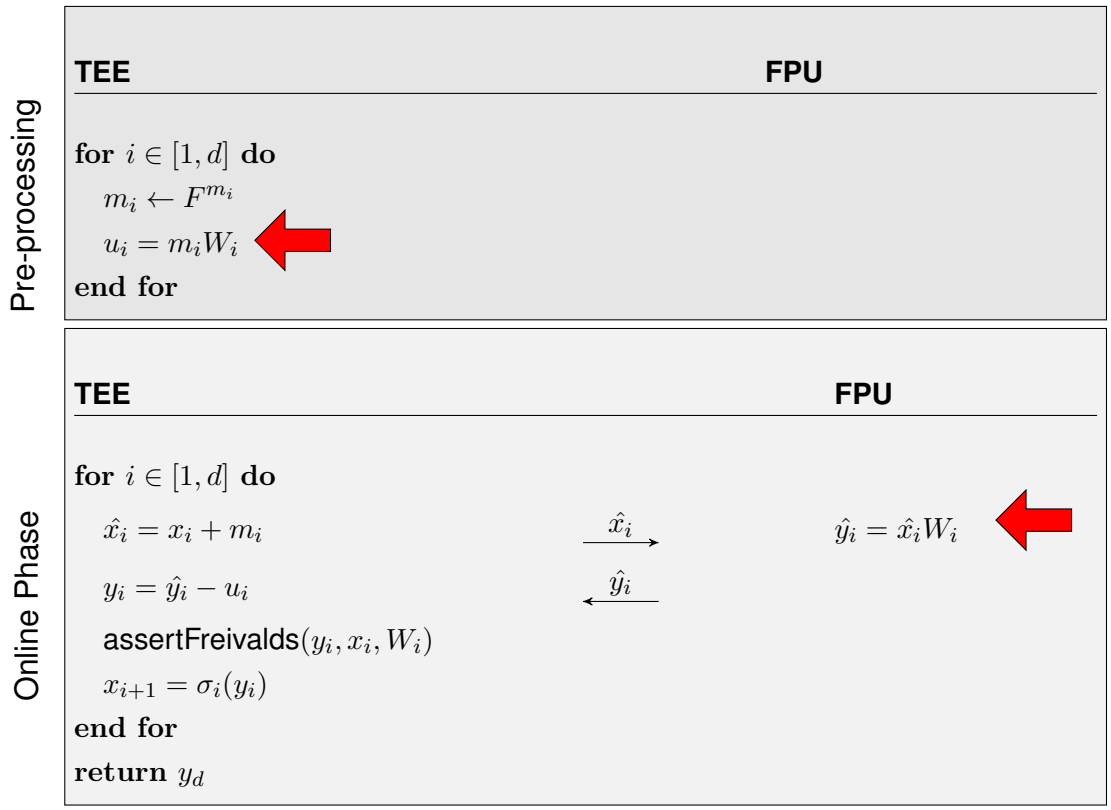

Figure 3.2: The SLALOM algorithm by Tramèr and Boneh [213]. Note that the marked calculations are essentially the same. Additionally, the FPU is idle during the preprocessing phase.

The TEE is used to perform the entire preprocessing phase, generating the random masks and calculating the corresponding unmasking values. The masks are random values, the unmasking values are the output of the individual network layers after inferring the network on the masks. Both masking and unmasking values need to be created for each layer individually, and, of course, a large pool of masks is required. The FPU is idle during the preprocessing phase. In the online phase, the client C sends their secret input x to the server S, where it is received and masked by the TEE. The masked values are then passed to the FPU, which calculates the linear layer. The masked result is passed back to the TEE, where the corresponding unmasking value is used to obtain the result of the linear layer. The algorithm ASSERTFREIVALDS is called to verify the FPU calculation. The following non-linear layer is then applied to this result, generating the input for the next linear layer. The new input is then masked, sent to the FPU and so on.

In a more formal way, the two phases work as follows:

Preprocessing Phase: For each layer NN_i of NN, the TEE generates a *masking value* $m_i \in \mathcal{X}_i$ uniformly drawn from $\mathcal{X}_i$ and computes the *unmasking value* $u_i = \mathsf{Lin}_i \cdot m_i$. The preprocessing phase is performed by the TEE, as it generates the masking values, which need to be hidden from the server owner and model owner.

Online Phase: To evaluate layer NN_i on secret input x_i, the TEE computes $\hat{\mathsf{x}}_i = \mathsf{x}_i + m_i$ and sends $\hat{\mathsf{x}}_i$ to the server. The FPU then computes $\hat{y}_i = \mathsf{Lin}_i(\hat{\mathsf{x}}_i)$ and sends $\hat{y}_i$ to the TEE. The TEE now computes $y = \hat{y}_i - u_i$ and sends $\mathsf{x}_{i+1} = \mathsf{NLin}_i(y_i)$.

Using this approach, one can show the input privacy of SLALOM by making use of the observation that the masking provides information-theoretic security.

Theorem 1 (Theorem 3.2 in [213])**.** *Assume that all random values are generated using a secure PRNG with security parameter λ. Then,* SLALOM *is a secure outsourcing scheme guaranteeing correctness and privacy.*

The approach taken by SLALOM works well if the offline phase is not included in the total runtime of the framework, and indeed SLALOM is considered one of the milestones in private inference and many future approaches rely on trusted hardware to build fast and secure solutions [85, 153, 170, 189, 223]. The SLALOM framework does, however, have some considerable weaknesses:

Total Runtime: In the SLALOM setting, *the entire network is evaluated on the masks* in the TEE during the preprocessing phase. As the masks have the same size as the input, the preprocessing phase plus the online phase take *more* time than just inferring the entire network on the TEE to begin with.

Implementation issues: Performing the preprocessing phase on TEEs may lead to memory issues as enclave size needs to be appointed before the enclave is created and swapping out the memory massively lowers the performance. In addition, it must already be known in the offline phase how many inputs need to be processed in the online phase for the masks precomputations.

Idle FPU: During the preprocessing phase, only the slow TEE works, while the much more powerful FPU is idle. As the server owner bears the cost of

generating the unmasking values in the TEE during the preprocessing phase, she has an inherent interest in an efficient use of resources.

Running out of Randomness: SLALOM is only suitable for restricted situations with sufficient time between the online phases that can be used to prepare sufficient preprocessing material. If the time between two online phases is too short, SLALOM will run out of randomness, leading to a large performance penalty.

It is unclear whether the authors were aware of these complications, as they did not implement the offline phase in the TEE in their PoC, and rather chose to do it insecurely on the FPU[1]. We would like to stress that none of these issues affects the correctness or security of SLALOM.

3.3 The CARNIVAL Primitive: Building OTPs from Public Randomness

In this section we formally define the cryptographic primitive behind CARNIVAL. In general, the goal is to let an untrusted entity evaluate the function $g(x) := g_\alpha(x)$ on a private input x and public (for both parties) parameters α. While we will discuss this task in a more general setting, it is instructive to first consider $g(x)$ as one layer of a neural network, i.e., $g(x) = \mathsf{NLin}(\mathsf{Lin}(x))$.

We will present two different approaches to select appropriate parameters. In the first approach, we show how to obtain a provable secure version of our primitive, based on the hardness of the Subset sum problem and the state-of-the-art attacks against it. However, choosing smaller parameters does not imply that our scheme is insecure. In the second approach, we thus present a scheme with much smaller parameters and present a simple experimental evaluation to better understand the security of these parameters.

[1] It is apparent that the authors knew about the insecure nature of their implementation from the code and reflected in the comment "This is obviously insecure, but we currently compute the unblinding factors outside of the enclave for simplicity" [212].

3.3.1 Masking Inputs

In order to keep x private, we first compute the value $h(x, k)$ that *masks* the value x via some secret key k. The untrusted party is then given $h(x, k)$, computes $g(h(x, k))$, and then returns this result to us. Finally, to obtain the desired result $g(x)$, we now need to retrieve it from $g(h(x, k))$.

We only consider masking functions $h(x, k)$ that use an *additive* masking, i. e. $h(x, k) = x + f(k)$ for some function f. Now, if $g(x)$ is linear, we have $g(h(x, k)) = g(x + f(k)) = g(x) + g(f(k))$. Hence, we only need to compute $g(f(k))$ to obtain $g(x)$. In the easier model of SLALOM, where the costs for preprocessing are ignored, we could compute $g(f(k))$ by ourselves in the preprocessing phase, as $f(k) = k$ here and the neural network described by g is public. But, by choosing $f(k)$ more carefully, we are able to reduce these preprocessing costs drastically.

Using Subset sum: Now, suppose that $f(k)$ is defined via a Subset sum problem. Recall that in Subsection 3.1.1, we defined the Subset sum problem over a finite field F_p as a set $S = \{s_1, \ldots, s_n\} \in F_p^n$ of n field elements and a function

$$f_{S,p}(k) := f_p(k, S) = \sum_i k_i s_i \bmod p$$

that maps a binary vector k to another field element. The parameters p and S (and therefore by definition also $n = |S|$) are public, while k is the private key. In order to facilitate the pseudo-randomness of the Subset sum problem, we will first sample the set $S = \{s_1, \ldots, s_n\}$ randomly from F_p. Note that this process can also be performed in public (as long as the random choice of S is guaranteed). Now, we can compute the unmasking set S_u with $u_i = g(s_i)$ for $i = 1, \ldots, n$ in public. We will discuss how to verify this computation in Section 3.5.

In the preprocessing phase, the client can choose a new random key k and compute $r = \sum_i k_i s_i \bmod p$ and $u = \sum_i k_i u_i \bmod p$. Note that the preprocessing phase thus only requires the addition of $2n$ field elements, but not computation of g.

In the online phase, the client only needs to calculate $\mathsf{x}+r$ and is given $g(\mathsf{x})+g(f(k))$, from which it subtracts u. Applying linearity of g again, we see that for $f = f_{S,p}$, we have

$$g(f(k)) = g(f_{S,p}(k)) = g\left(\sum_i k_i S_i \bmod p\right) = \sum_i k_i g(s_i) \bmod p = u,$$

as k_i is only a binary scalar. Hence, during the online phase, the client is able to compute $g(\mathrm{x})$ using only two additions.

One of the main advantages of our approach is the fact that we can use the same knapsack instance, generated during the setup phase, for a large number of preprocessing and online phases. We only need to guarantee that the preprocessing phase produces a fresh key. Hence, a single setup with n items can be securely used for roughly $2^{n/2}$ preprocessing and online phases (at which point a key will probably repeat).

3.3.2 Provable Security

In this subsection, we will show that our approach is provably secure as long as the underlying Subset sum problem is sufficiently hard. Clearly, there are two parameters that influence the security of the Subset sum problem and thus of our approach. The first parameter is the size of the individual field elements. In our scenario, this size is fixed by the size of the elements of the neural network. Hence, we will focus our attention on the second parameter, the number n of different items S_i in the Subset sum instance. For performance reasons, it is desirable to keep n as small as possible without compromising the security of the system.

We will denote the security parameter by λ and abbreviate the term probabilistic polynomial time by PPT. A sequence $\{p_\lambda\}_\lambda$ is *negligible* if, for all $c \in \mathbb{R}_{>0}$, there is $\lambda_0 = \lambda_0(c)$ such that $p_\lambda < 1/\lambda^c$ for all $\lambda \geq \lambda_0$, i.e., it is smaller than the inverse of every polynomial. We call a function ensemble $\{f_\lambda\}_\lambda$ with $f_\lambda \colon D_\lambda \to R_\lambda$ to be *one-way*, if $f_\lambda(x)$ is computable in polynomial time and

$$\Pr_{x \leftarrow D_\lambda}\left[f_\lambda(\mathcal{A}(f_\lambda(x))) = f_\lambda(x)\right]$$

is negligible for all PPT attackers $\mathcal{A}$. Hence, an attacker that is given the value $f_\lambda(x)$ should not be able to construct a value x' (not necessarily identical to x) such that $f_\lambda(x') = f_\lambda(x)$. Finally, we call such a function ensemble $\{f_\lambda\}_\lambda$ a *pseudo-random generator*, if $f_\lambda(x)$ is computable in polynomial time, $|R_\lambda| > |D_\lambda|$, and if

$$|Pr_{x \leftarrow D_\lambda}[\mathcal{A}(f_\lambda(x)) = 1] - Pr_{y \leftarrow R_\lambda}[\mathcal{A}(y) = 1]|$$

is negligible for all PPT attackers $\mathcal{A}$: An attacker that is either given a completely

random element from R_λ or $f_\lambda(x)$ for a completely random element from the smaller set D_λ should not be able to distinguish between these scenarios.

From a purely asymptotic point of view, there is a very close relation between the one-wayness of the knapsack function and its pseudo-randomness, already established by Impagliazzo and Naor [94].

Theorem 2 (Theorem 2.2 in [94])**.** *If the Subset sum function ensemble* $\{f_p\}_p$ *with* $f_p \colon F_p^n \times \{0,1\}^n \to F_p^n$ *with* $\log(p) \geq (1+\epsilon)n$ *for some* $\epsilon > 0$ *is one-way, then it is also a pseudo-random generator*[2]*.*

Theorem 2 is already sufficient to prove the security of CARNIVAL, following the security definitions of Tramèr and Boneh.

Theorem 3. *Let* CARNIVAL *be the above protocol to compute a linear function* g*. Assume that all random values are generated using a secure PRNG with security parameter* λ *and that the knapsack function* $f = f_{p,S}$ *is a one-way function chosen with security parameter* λ*. Then,* CARNIVAL *is a secure outsourcing scheme guaranteeing correctness and privacy.*

Proof. The correctness follows easily from our discussion above. Theorem 2 implies that the one-wayness of f also guarantees pseudo-randomness. Then, the pseudo-randomness of f means that we can replace $f(k)$ by a completely random value r. The runs of $\texttt{Online}_{\mathsf{C},\mathsf{S}^*}(\texttt{param}, \texttt{state}, \mathsf{x} + r)$ and $\texttt{Online}_{\mathsf{C},\mathsf{S}^*}(\texttt{param}, \texttt{state}, \mathsf{x}' + r)$ are identical, so privacy is guaranteed. □

Example: Outsourced Matrix Multiplication We illustrate the primitive with the following scenario: A client wants to multiply a private vector x with a matrix W. Since this is computationally expensive, the client would like to outsource the matrix multiplication to a server. The input vector should of course remain private. In this scenario, $g(x) = x \cdot W$. Since $g(x+m) = g(x) + g(m)$ it is possible to let the server generate many random masking vectors m_j and calculate the corresponding $g(m_j)$. The masks m_i are then used as S to generate the OTP r and the unmasking value u for the OTP is calculated accordingly.

[2] As [94] does not consider one-way functions with public parameters, the function should also output S, but we ignore this here for the sake of readability.

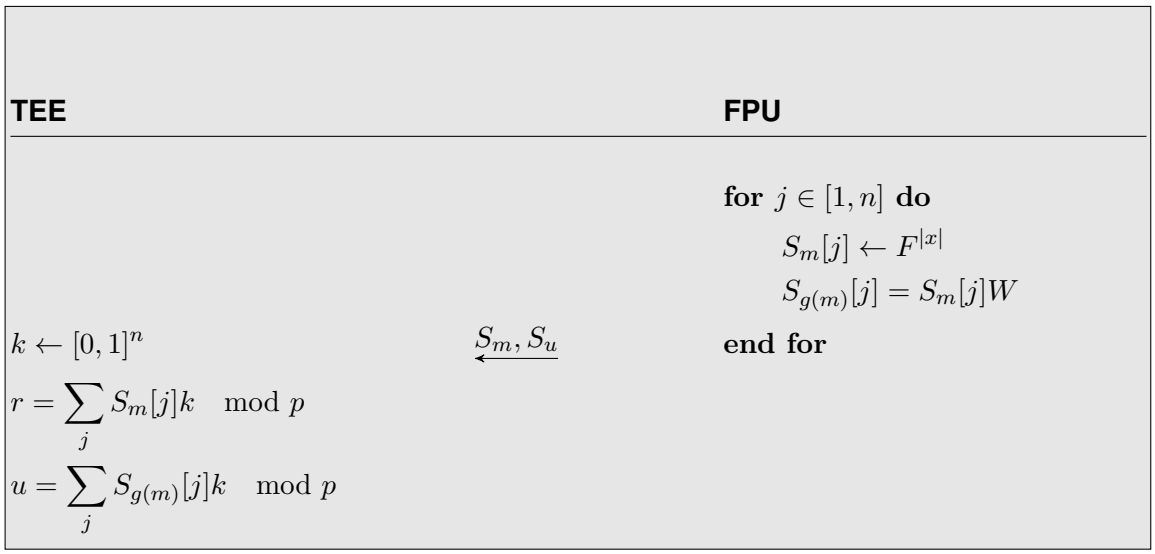

Figure 3.3: Illustration of the setup phase and one corresponding preprocessing phase in CARNIVAL: Note that the FPU is generating most of the randomness and the communication is unilateral. Furthermore, a single setup phase can correspond to many preprocessing phases. One preprocessing phase is required for each of the b expected batches, but they can all be computed ahead of the online phase.

Figure 3.3 shows the entire preprocessing phase including outsourced generation of a set S_m of masking and a corresponding set S_u of unmasking values by the server and the required processing step by the client to obtain the secret OTP for masking and unmasking. Note that the client only has to perform additions. After the preprocessing, the client would add an OTP to the input value x and send the masked input to the server. Once the client receives the masked result, it subtracts the corresponding unmasking value to obtain the result.

Choice of Parameters While the above result already implies the security of our construction, it only provides an asymptotic bound that might not be meaningful for smaller, practically relevant parameters. Furthermore, the reduction from one-wayness to pseudo-randomness relies on the Goldreich-Levin theorem [72], which has a large run time and is thus probably not tight.

In the following, we thus discuss the choice of practically relevant security parameters that take the best known attacks into consideration. To the best of our knowledge, the current state-of-the-art attack against the Subset sum problem was

developed by Bonnetain et al. and runs in time $\tilde{O}(2^{0.283n})$ for field size $p \approx 2^n$ [20]. Hence, in order to obtain 128-bit security, by using Theorem 2, we need $n \geq 453$. Using this parameter, we obtain a cryptographically secure protocol.

Result: We obtain a cryptographically secure protocol by using $n \geq 453$ knapsack elements.

We stress here that using smaller parameters of n only show that the attack of Bonnetain et al. [20] is able to break the one-wayness of the Subset sum problem. However, the distribution generated by the Subset sum instance is still very close to uniform. Consider, e.g., the field elements $S = \{s_0, \ldots, s_{n-1}\}$ with $s_i = 2^i$ and $p \approx 2^n$. For a random key choice k, the value $f_{S,p}(k)$ is clearly uniformly distributed, although the underlying Subset sum problem is trivial (as the solution can be directly deduced from the single bits of $f_{S,p}(k)$).

3.3.3 Experimental Security

We distinguish between two types of security: The *cryptographic security* is provided by a key length generally believed to be secure against a probabilistic polynomial time attacker. For the Subset sum problem, $(1/0.283)n$ elements in the set S directly translate to n bits of security. Our above discussion already guarantees security in this case for sufficiently large parameters n and p.

However, as discussed above, smaller parameter sets do not directly imply insecurity of our approach. We thus also consider another type of security, called *statistical security*. The statistically secure key length is the key length where the distribution of the possible sums is statistically indistinguishable from a uniform distribution. An example can be seen in Figure 3.4. In order to understand the parameter landscape for smaller, practically relevant parameters (such as $p = 2^{128}$), we performed some preliminary experiments to answer the following question:

> How low can n be so that $f_{(S,p)}(k)$ is indistinguishable from the uniform distribution within an acceptable error margin?

We conducted experiments to determine possible boundaries for n: For increasing values of l, we ran tests on 100 randomly generated sets S with $n \in [2l-4, .., 2l+7]$. For each set, we calculated all possible sums and checked total variation distance, also referred to as statistical distance, to the uniform distribution. We decided that

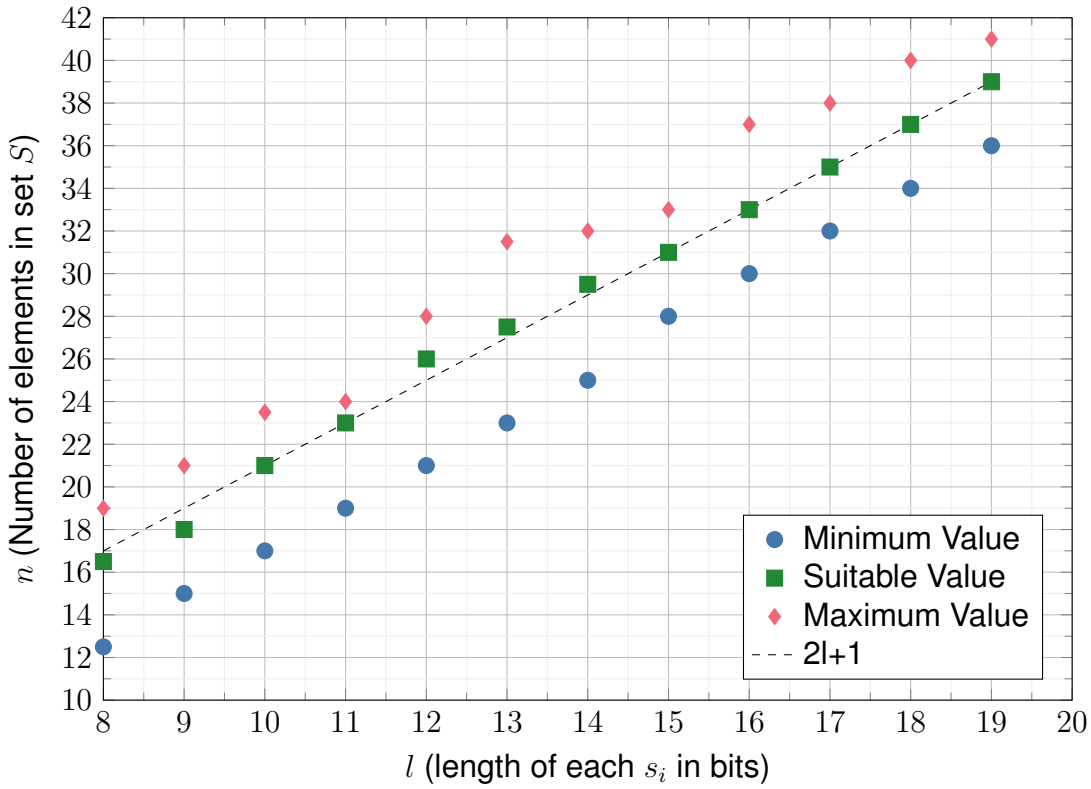

Figure 3.4: The minimal, suitable and maximal number of elements in a set S containing elements of size l to ensure a distribution of sums that is statistically indistinguishable from a uniform distribution: The minimum value denotes the lowest set size that produced a uniform distribution, the suitable value the set size that was indistinguishable within 95% probability, and the maximum value the set size that always generated uniformly distributed sums.

an n was suitable if the chance of obtaining a distribution indistinguishable from the uniform distribution was at least 95%. The results are displayed in Figure 3.4. We stress here, that these experiments should not be treated as a replacement for a sound security analysis, as we only consider relatively small parameters and do not have the necessary amount of samples to truly derive statistical guarantees. These experiments thus only indicate that smaller parameter sizes might be possible for our application.

Looking at the results for lower values of l up to $l = 12$, the results seem to grow linearly and indicate an optimal value of n can be found for $n \approx f(l) = 2l + 1$ for higher values. With the results above, it could now be shown experimentally

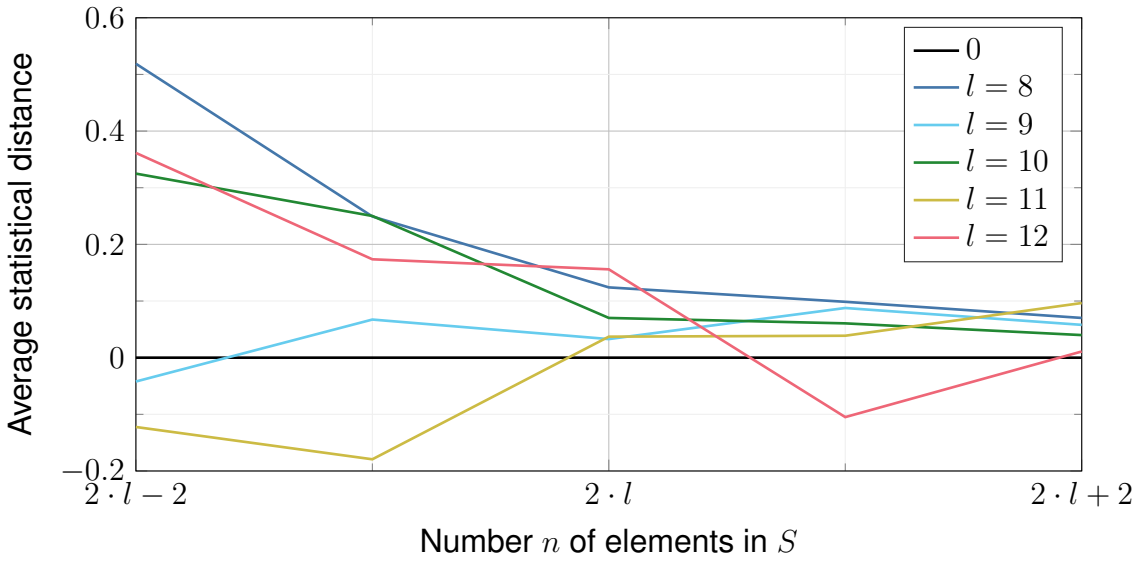

Figure 3.5: Test results for the selection of the possible set value range. The y-axis shows the percentage of restricted sets closer to a uniform distribution than the full sets. Results above the black lines display better results for the restricted range $s_i \in \{2^{l-1}, p-1\}$.

that the key length can be dramatically smaller than the key length required for cryptographic security: To obtain a distribution indistinguishable from a uniform distribution with a certainty of 95%, it is shown experimentally, that the key length needed for this is only $|k| = n = 17$ for $l = 8$. In the case of the cryptographic key length, the value for $l = 8$ would be $n = 30$. Finally, our experiments have shown that the security of the system for $l = 8$ would no longer increase significantly at $n = 17$ if the key length were increased, for example to $n = 30$.

Parameters We chose $p = 2^l$ for the ring size following [94] as it is a value that is simple to obtain (as opposed to primes) and allows the use of bitshifts in the calculation. Additionally, we checked whether restricting the possible elements in s_i to $s_i \in [x, p-1]$ with $0 < x \ll p$ instead of using $s_i \in [0, p-1]$ leads to a significant change in the sums that can be obtained from the set S.

We ran experiments for $x = (p-1)/2$. By calculating all possible sums for 600 random sets, we could determine the average statistical distance between the regular range with $s_i \in [0, p-1]$ and the restricted range with $s_i \in [2^{l-1}, p-1]$ for each combination of l and n. The results are displayed in Figure 3.5.

The graph displays the percentage of test cases for each combination of l and

n where the restriction of the set elements to $s_i \in [2^{l-1}, p-1]$ had better results than using set elements from $s_i \in [0, p-1]$. Every result above 50% means that the restricted set was closer to a uniform distribution than the full set. Since the restricted range resulted in a significant improvement, the range was limited to $s_i \in [2^{l-1}, p-1]$.

3.4 SLALOM at the Carnival: Integration in SLALOM Framework

We consider an inference-as-a-service scenario where clients send inputs and get the corresponding results. The input x and the result $\mathrm{NN}(\mathrm{x})$ are supposed to stay private while the machine learning model NN consisting of the network architecture, the weights W and the activation functions σ are known to the FPU. In this scenario, the client has to cover the cost of the precomputation phase. Thus, an expensive offline phase will directly result in cost for the client.

As shown in Figure 3.6, in the original SLALOM the TEE masks the input values with random numbers, sends the blinded values to the FPU and unblinds the result of every layer with pre-calculated unmasking values: for each weight matrix W_i, a random vector m_i is sampled and $u_i = m_i W_i$ is calculated in the TEE. The overall effort for the offline- and online-phase is thus higher than just inferring the entire model in the TEE. While splitting the computational performance analysis into an offline- and an online phase seems to be common practice in machine learning settings, a costly offline phase is still a relevant factor in many settings.

A second problem with the SLALOM approach is that the number of unblinding factors computed in the offline phase is limited, which directly implies that the system can run out of unblinding factors in the online phase. The IaaS provider would then have to shut down the system and launch a new offline phase, which is costly in both time and money.

CARNIVAL can be used to overcome both of these challenges: It uses the FPU to generate many potential masks and the corresponding unmasking values. We thus incorporate CARNIVAL into the SLALOM framework, which generates a new framework: SLALOM at the CARNIVAL, or S@C. With appropriate scaling and buffering, the setup phase on the FPU can be parallelized for several layers, increasing

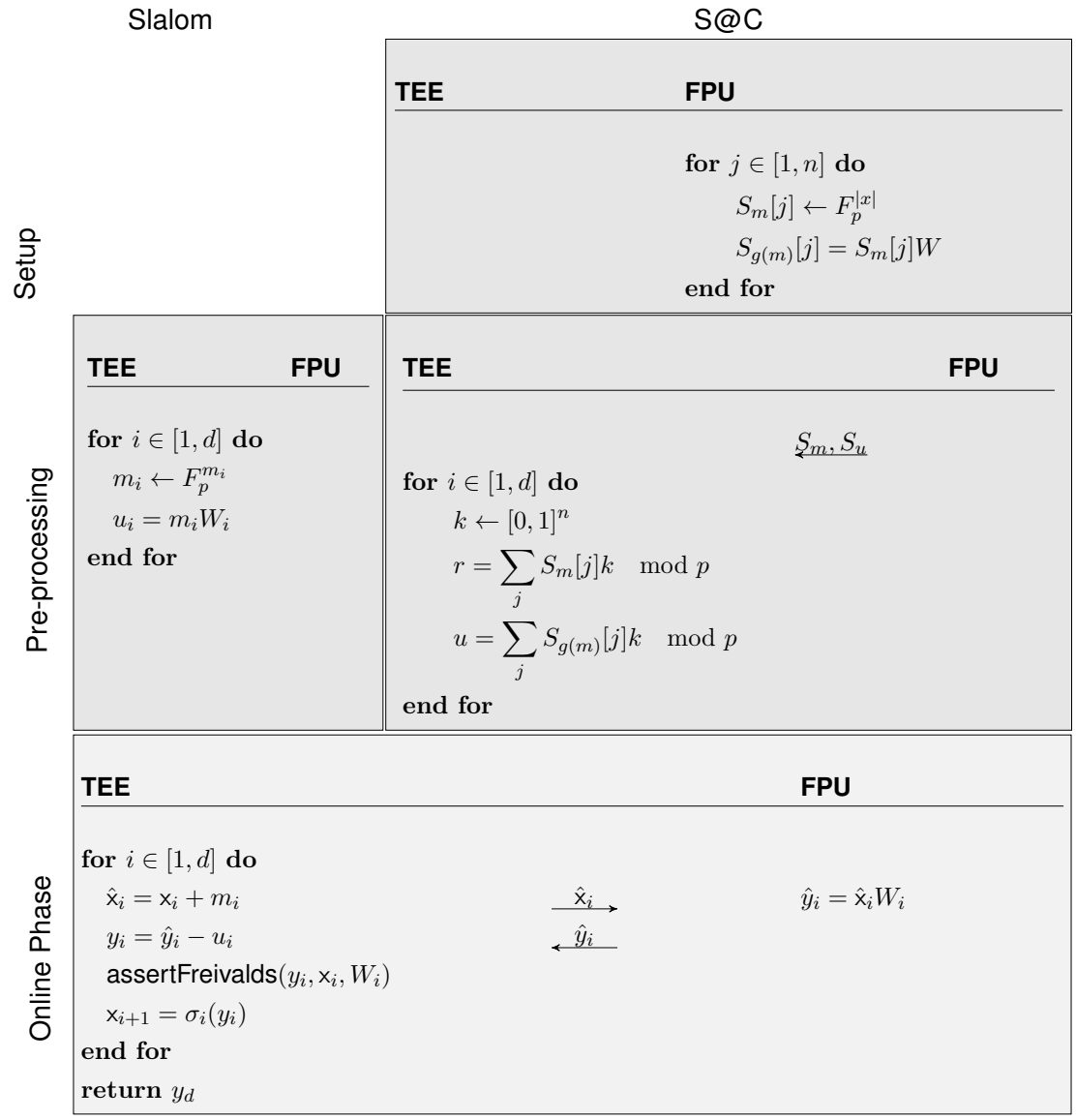

Figure 3.6: SLALOM pseudocode for both variants for a network with d layers. While the preprocessing phase of our variant is longer, it significantly reduces the computational load of the TEE by outsourcing more work to the FPU. The online phase is the same in both protocols except the assertFreivalds step is optional in S@C if integrity is not a security goal.

the performance further. With the public set S_m of possible masks, the TEE can then pick a random key k to add up some of the masks and generate a secret one. The corresponding public unmasking values in set S_u need to be summed up as

well. As shown in the previous sections, a polynomial time attacker has no efficient way of telling which of the provided masks were used to build the final mask, and thus cannot determine the private user input due to the pseudo-randomness of the Subset sum problem. The particular property of the CARNIVAL scheme leveraged here is the fact that linear transformations on the elements of set S lead to linearly transformed OTPs, allowing a lot of formerly secret computations to be moved to the FPU.

The masks used in SLALOM are 32-bit integers. In line with the experimental results from Subsection 3.3.3, we would require $n = 2l + 1 = 65$ elements to achieve statistical security.

Result: We achieve statistical security with $n \geq 2l + 1$ elements, which corresponds to 65 elements in the concrete case of the SLALOM framework.

3.4.1 Performance Analysis and Comparison

The complexity of a conventional 2D convolution is quadratic with three hyperparameters: number of channels C, kernel size K, and spatial dimensions of the input H and W, and its computational complexity is $\mathcal{O}(C^2K^2HW)$. An addition of a matrix of the same dimensions can be performed in $\mathcal{O}(CKHW)$. In S@C, the TEE needs to generate one random n bit binary number and then perform a maximum of n additions followed by modular reductions. The expensive operations of convolutions and the generation of large amounts of random numbers are shifted to the FPU, and the workload of the much slower TEE is reduced by orders of magnitude. Assuming that the FPU is f times faster than the TEE, a performance gain is achieved if $n < f$. Since n is 453 in the cryptographically secure scenario, S@C is faster than SLALOM if $f > 453$ or, for the statistically secure scenario, if $f > 2l + 1$.

As mentioned in the introduction, there is no implementation of SLALOM available for comparison. To nevertheless provide a performance comparison, we implemented a benchmark consisting of a full matrix multiplication. We chose the benchmark since ML networks usually contain convolutions or matrix multiplications in their linear layers, matrix multiplications being the faster of the two operations. We thus compare to the harder case. As we are looking for a lower bond, we use the proveable secure variant. We ran the benchmark on a GPU (Nvidia H100) and within Intel SGX (2x Intel Xeon Gold 6438Y+).

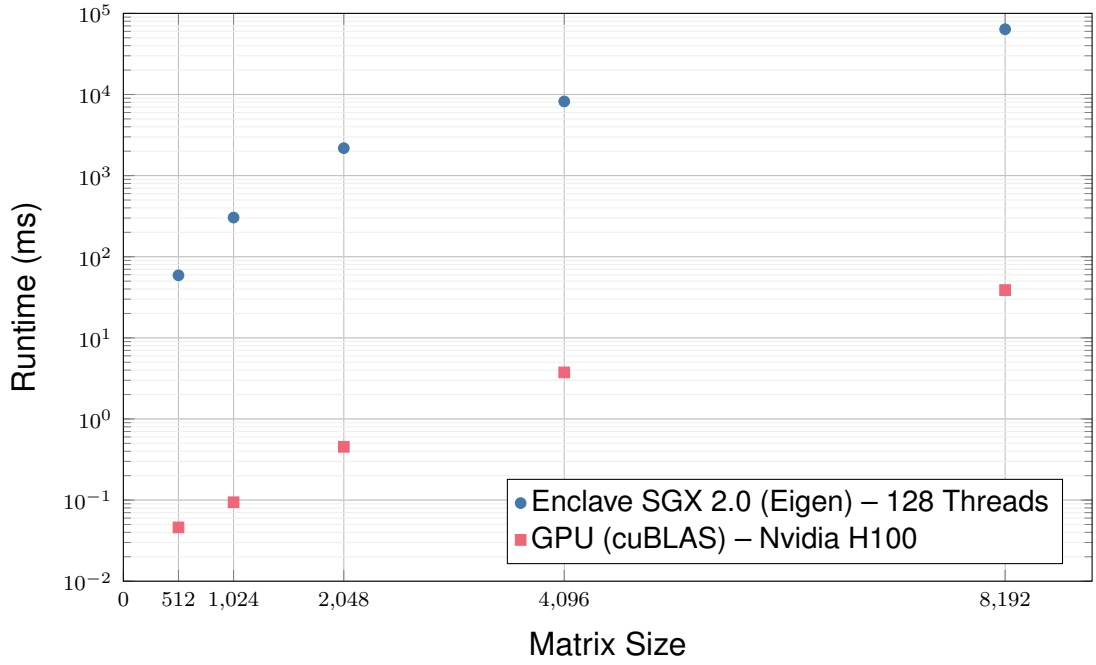

Figure 3.7: Runtime comparison for a matrix multiplication within the enclave and on the GPU. The C++ library EIGEN was used within the enclave and the cuBLAS library on the GPU

The runtime results are depicted in Figure 3.7. As expected, the enclave is the slowest option regardless of matrix size, while the GPU is the fastest. We can see that the speedup when using the GPU as opposed to the TEE is three orders of magnitude regardless of the matrix size and ranges between $f = 1\,000$ and $f = 5\,000$. That means that we always have a speedup of over $1\,000$, meaning that $f > 1\,000$. In the provable secure scenario, CARNIVAL's preprocessing phase will thus be between 2.2 and 11 times faster than SLALOM's.

Result: Compared to SLALOM, CARNIVAL achieves a speedup of 2.2 up to 11 in the provable secure variant. In the experimentally secure variant, the speedup is between 15.4 and 76.9.

We can thus provide both statistical and cryptographic security while simultaneously gaining at least an order of magnitude in performance.

3.5 Integrity Add-on

While we have a TEE that we trust, the FPU is an untrusted component. It is thus possible that it cheats at any point of the framework. The client can ascertain the FPU's honesty at three points:

1. During the setup phase, when the random set elements $S_i = m_i$ are generated.
2. During the preprocessing phase, when the unmasking values u_i are calculated.
3. During the online phase, when the results of the linear layers are returned to the TEE.

SLALOM uses an error term of $t = 2^k$ and achieves t-integrity by k repetitions of Freivalds' algorithm per layer, as shown in the following theorem.

Theorem 4 (Theorem 3.2 in [213])**.** *Assume that all random values are generated using a secure PRNG with security parameter λ. Then* SLALOM *provides t-integrity for $t = 2^k$ if Freivalds' algorithm is repeated k times per layer.*

As we will see, the same holds for CARNIVAL.

3.5.1 Detect FPU Cheating During Set Generation

The TEE can detect a dishonest FPU in this case if a PRNG with leap ahead property is used. The leap ahead property allows the TEE to request specific values from the PRNG, for example the first, fifteenth and eighty-third random value [234]. The TEE can use the PRNG to generate k random values from S itself and compare them to the ones sent by the FPU. The leap ahaead property ensures that the TEE does not have to generate all intermediate random values, which would void the performance gain. The hardest case to detect would be the FPU cheating on exactly one generated value. Since n values are generated, the probability of not detecting the one the FPU cheated on when randomly reproducing k values is $\approx 1 - \frac{k}{n}$. The detection probability obviously increases drastically if the FPU cheats more than once.

3.5.2 Detect FPU Sending Dishonest Unmasking Values

Assuming that the FPU generated the masking values fairly, we can apply Freivalds' algorithm to ensure it also calculated and sent honest unmasking values [64]. The Freivalds' test is a probabilistic test with one-sided error that checks whether two $j \times j$ matrices were multiplied correctly without comparing all the elements individually. To verify the unmasking values, the following steps are taken. n set elements from the masking set are aggregated into an $n \times n$ matrix M and the corresponding n unmasking vectors into an $n \times n$ matrix U. To check whether the multiplication were performed correctly, we sample a random binary vector t of length n and calculate

$$W \cdot \left(t^T M\right) - t^T U = 0.$$

If we repeat this k times, the error term is bound by $\mathsf{e} \leq \left(\frac{1}{2^{2k}}\right)$ since we use binary vectors. To achieve the desired error term of 2^{-40}, we need $k \geq 20$.

3.5.3 Detect FPU Cheating During Inference

Ascertaining whether the FPU sent an honest inference result is similar to detecting whether it cheated in creating the unmasking values. Here we build batches from the masked input values instead of the set elements, and compare them with the inference results. We calculate

$$W \cdot \left(t^T (M + x) t\right) - t^T (\hat{y} - U) = 0.$$

We sample a random binary vector t of length n, so to achieve an error term $\mathsf{e} \leq 2^{-40}$ similar to SLALOM, we also require at least 20 repetitions.

As opposed to SLALOM, we only use random numbers from $\{0, 1\}^n$. We can thus perform Freivalds' check without relaxing our assumptions about the TEE: Since we use a binary vector to detect the error, we can still calculate everything using only additions and modular reductions, at the cost of having to perform about 10 times more tests.

Equipped with this integrity test, we can conclude our main theorem.

Theorem 5. *Let* CARNIVAL *be the above protocol to compute* NN *with the integrity test. Assume that all random values are generated using a secure PRNG with*

security parameter λ and that the knapsack function $f = f_{p,S}$ is a one-way function chosen with security parameter λ. Then, CARNIVAL *is a secure outsourcing scheme for* NN *guaranteeing correctness, privacy, and t-integrity for $t = 2^k$ if Freivalds' algorithm is repeated k times per layer.*

3.5.4 Another Way Forward: DASH

While working with SLALOM, we conducted a thorough analysis of existing outsourced computation schemes and the various techniques and hardware in them. In an alternative direction to CARNIVAL, Jonas Sander, Sebastian Berndt, Thomas Eisenbarth and I developed DASH. DASH stays within the same attacker and hardware model of a TEE and a GPU on an untrusted server. Unlike CARNIVAL, DASH does not rely on masking, but on the MPC technique of *Garbled Circuits (GCs)*. Binary garbled circuits proved to massively inflate communication between the protocol parties.

In 2016, Ball et al. generalized the concept of GCs to *arithmetic GCs* [13]. These work with arithmetic operations such as addition, subtraction and multiplication and are both more efficient and easier to handle than binary GCs. In 2019, they reformed them to accommodate ANN-specific arithmetic [12], setting a new milestone in GCs for machine learning. DASH builds on the cryptographic building blocks of these works. We extend them with efficient power-of-two scaling in the Chinese remainder theorem representation. Just as SLALOM and CARNIVAL, DASH uses an offline-online approach and leverages the GPU. DASH is the first framework using GPU acceleration for arithmetic frameworks and outperforms previous state-of-the-art for outsourced inference using GCs.

The workflow of DASH is illustrated in Figure 3.8. For every inference, DASH performs the following steps:

1. The model owner sends the model NN to the garbling device.
2. The garbling device creates the GC gNN from the model, along with the encoding information e and decoding information d.
3. The TEE sends the GC gNN to the inference device, in this case the FPU. The offline phase is over now.
4. The input owner sends the input data $\mathrm{x} \in \mathcal{X}$ to the garbling device.

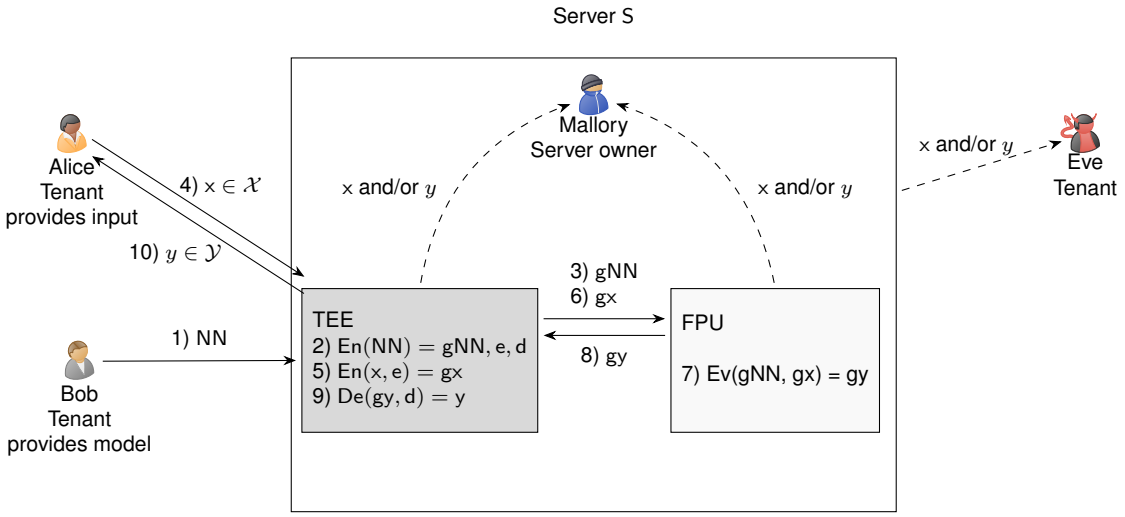

Figure 3.8: The workflow of DASH: Mallory host a server S. Bob owns a trained model NN and sends it to the server to offer inference as a service. Alice is an honest tenant and sends input x to receive output y. Both Eve, an attacking tenant, and Mallory try to extract information about x and y. As the CPU and LLC are not relevant for our scenario, they are omitted from the setup.

5. The garbling device then creates the garbled input gx from x and e.
6. It sends gx to the inference device.
7. The inference device evaluates gNN to obtain the garbled output gy.
8. The garbled output gy is sent back to the garbling device.
9. The garbling device creates the output y from gy and d.
10. The output y is sent to the output owner, which is equal to the input owner in our case.

Of course the garbling device, in our case the TEE, attests against input owner, output owner and model owner as they enter the process.

The similarities and differences of CARNIVAL and DASH are collected in Table 3.1. Both SLALOM and CARNIVAL accelerate the encrypted linear layers on the GPU, but rely on a co-located TEE for the non-linear layers. As the entire input for each layer needs to be sent to the other component respectively, the co-location of TEE and GPU is mandatory if a communication bottleneck should be avoided. DASH can also use a TEE, but only needs to communicate the model in- and outputs

Table 3.1: Comparison of the properties of CARNIVAL and DASH, two frameworks building on SLALOM.

Property	CARNIVAL	DASH
Security goal	Input privacy	Input and Model Privacy
Performance focus	Preprocessing phase	Online phase
Linear layers	Masked on FPU	GCs on FPU
Non-linear layers	TEE	GCs on FPU
Requires TEE	Yes	No, but profits from it

with the inference device, resulting in a non-interactive online phase. The entire NN can be garbled securely in the offline phase. DASH thus manages to combine the advantages of GCs, namely low communication complexity, with the massive parallelism of a GPU. In addition, our solution has a constant memory requirement independent of the number of input providers.

While CARNIVAL aimed at optimizing this offline phase, DASH purely strives to implement an efficient online phase using GCs. Unfortunately, the two approaches cannot be combined easily, as mixing GC and masking techniques leads to massive overhead in the conversion between the linear and non-linear layers [14, 15, 37, 140, 152, 159, 160, 174, 185]. Speeding up the offline phase of DASH may be a good direction for future work.

3.6 Related Work

Securing outsourced ML inference is a very active research area and many high-quality SoK papers have been published recently [38, 107, 154, 215]. We focus on direct follow-up works of SLALOM and efforts to improve preprocessing. For a wider overview on this drastically growing field, we refer to the papers mentioned in Section 2.2 and to surveys such as [59, 141, 183, 209]. Many outsourced ML approaches divide their calculations into an offline preprocessing phase and an online phase. However, the preprocessing phase is often not included in the performance analysis [189, 226] or optimization efforts: Authors argue that it can be

performed offline ahead of time and thus does not impact the total time required for the inference. An exception are the works MUSE, DELPHI, GAZELLE and SIMC [36, 110, 129, 152], which all use additive HE for the linear layers and binary GCs for the non-linear layers. The bottleneck of this architecture is the conversion between the two, which requires extensive amounts of communication and calculation. The earliest work is GAZELLE, which has roughly 60% of its runtime and 80% of its communication in the offline phase. With DELPHI, Mishra et al. showed that they can push 99% of their computational load to the input independent preprocessing phase and also reduce the overall cost of a (deep) network by approximating the activation functions instead of using ReLUs. Their goal is input privacy. MUSE builds on DELPHI, but focuses on the security goal of model privacy, leading to worse runtimes in both preprocessing and online phase. The HE and authenticated Beaver triples used in the non-linear layers of MUSE replaced by oblivious transfer and OTP encryptions in SIMC, leading to a significant performance and communication gain in the preprocessing phase while maintaining the same values in the online phase.

It is worth mentioning that speeding up the preprocessing of outsourced computation is a field of research independent of machine learning, and efforts to enhance the performance and communication of preprocessing steps are, of course, present in SMPC tasks. Scholl et al. provide a good overview on this topic [191]. To the best of our knowledge, there has been no prior attempt to speedup the preprocessing phase of SLALOM.

SLALOM uses a combination of TEE and masking to process confidential values on an insecure GPU. It only supports inference, as the whole preprocessing phase relies on fixed model parameters, which are not given during training. With DARKNIGHT, [85] directly follow up on this issue of SLALOM and add another wrapper to enable learning as well. They do, however, leave the existing framework as it is and do not alter the preprocessing phase [85]. We thus expect that our performance improvements could also be used by DARKNIGHT. Incorporating S@C would improve the overall performance of DARKNIGHT.

As mentioned in Section 2.2, SMPC is an alternative approach to the cryptographic approach CARNIVAL uses. Mohassel and Zhang [160] developed SecureML in 2017, where two untrusted servers train a model using 2-party MPC techniques. Just as the HE approaches, they approximate the activation functions sigmoid and softmax to reduce complexity, but they also use GCs to further speed up these

functions. ABY^3 [159] is a variant of SecureML for three servers. The scheme Minionn combines secret sharing with GCs in a one server setting, again with approximated activation functions. ABY^3, Minionn and Deepsecure [188], which uses binary circuits for garbling and oblivious transfer for non-linear layers, work on networks with few layers, but suffer from the huge overhead each activation layer adds. In the end, they can all only work with networks with 3 layers or less. Gazelle improved on this boundary by using HE in the linear layers and GCs in the activation layers [110] while optimizing packaging and encryption between the layers. The techniques from this scheme were adapted to the GPU by the Delphi scheme [152]. The HE computations are moved to an offline phase again, which is also included in the performance analysis. The offline phase uses up the vast majority of the runtime.

DASH also falls into the category of SMPC schemes using GCs. Just as Piranha in 2022, DASH adds usability as a distinguishing feature as well as performance [189, 226]. Dash allows both model owners and clients a low-threshold entry to efficient garbled circuits by enabling model loading from standard files, without deep knowledge of GCs. The Piranha platform follows a similar approach and enables developers of secret-sharing-based MPC schemes to leverage a GPU without knowledge of GPU programming. In both cases models do not need to be retrained.

3.7 Considerations and Potential Attacks

We tried to build on the code provided by Tramer and Boneh, but found strong obstacles very soon: The provided implementation does not actually implement the offline phase described in the paper, and implementing the offline phase as described leads to segmentation faults due to limited enclave memory. While newer enclaves provide more memory, we would have to re-implement the entire SLALOM framework on current libraries, programming languages and hardware. We thus do not provide an implementation of S@C, just the benchmarks provided in Subsection 3.4.1. Implementing a maintainable version of SLALOM and S@C, maybe even for practical use, is worth considering.

CARNIVAL does not tackle an entirely new problem. It improves the offline phase of a previously introduced algorithm. It does not help with the large open problem of

privacy preserving learning. While it is possible to apply the CARNIVAL primitive to other scenarios, we only used it to design the S@C framework.

Just like SLALOM, S@C requires communication between the TEE and the FPU between each linear/non-linear layer. As a result, the TEE and the FPU have to be co-located to avoid a communication bottleneck. While it is possible to use the CARNIVAL primitive without a TEE, e.g. on a machine under the control of the client, it cannot be applied to machine learning inference without a co-located TEE.

We discuss the security of CARNIVAL in Subsection 3.3.2 and Subsection 3.3.3. It is however worth mentioning that we base our security strongly on the TEE assumption. If the TEE is insecure, the security of SLALOM and CARNIVAL is void. One potential attack class that is often mentioned in the context of TEEs are *side channel attacks*. As these are *implementation* attacks, we stand by the TEE assumption: The fact that there are leaky implementations of algorithms that run within a TEE does not mean that TEEs are insecure. As we did not completely implement CARNIVAL, we also did not test it for side channel resistance. As the framework only contains secret additions and modular reductions, it is save to state that it is possible to implement the protocol without side channel leakage.

3.8 Summary and Conclusion

With CARNIVAL, we developed a way to combine public knowledge in order to obtain secret keys. By applying the Subset sum problem over finite fields, we can combine pre-computed masks and unmasking values to generate secret masks. We provide values for both provably secure and experimentally secure settings.

By incorporating CARNIVAL into SLALOM, we generate a new framework S@C. It is used to split the workload of machine learning inference safely between a TEE and an FPU. The linear layers are evaluated on the FPU on masked values, while the non-linear layers are evaluated within the TEE. Since we assume a co-located TEE and FPU, the communication cost is negligible. The model does not need to be retrained or adapted in any other way. However, as the masking requires quantization, a small loss in accuracy is to be expected when comparing to the native model.

S@C provides the same security guarantees as SLALOM, but includes an efficient preprocessing phase. We can leverage the existing resources better as we use the FPU for a larger portion of the framework. Our benchmark experiments, matrix multiplication and convolution, show that S@C would speedup the preprocessing phase of SLALOM by at least factor two in the provably secure scenario. If experimental security is enough, the speedup would be much higher.

CARNIVAL can be used in any setting where a weak, trusted device and a powerful, untrusted device compute a linear function. In that case, additive masking can be applied. Several other followup works for SLALOM stay within this scenario and could thus benefit from CARNIVAL. With their approach Origami, Narra et al. stay very close to SLALOM, but offload the remaining layers completely and in an unprotected manner to the FPU after a certain threshold [167]. Still, as the first half of their algorithm is just like SLALOM, it would benefit from S@C. Wu et al. offer the user to choose which layers will be evaluated with enhanced privacy by introducing *privacy levels*. While their approach currently only offers a choice to evaluate with the TEE or out of it, using masking on individual layers could be used as additional privacy level [232].

In addition to CARNIVAL, we present DASH, a GC based framework in the same setting as SLALOM. It uses LabelTensors to ensure the inherent parallelism of neural networks can be leveraged, for example by a GPU, despite the GC approach. In addition to an improvement over state-of-the-art outsourced inference, DASH offers a user-friendly loading mechanism for models, which can help to make secure ML solutions accessible to a wider audience.

4

Spoiler: Boosting Attacks with Secret Address Information

With CARNIVAL we now have a scheme that helps to protect data on a shared machine cryptographically on one side by applying masks. On the other side, it relies on trusted execution environments. One class of attack a TEE does not protect from are *side channel attacks*. In this chapter, we show how effects created by one process on the shared resources can be measured by an attacker process using the same resource. These effects can then be used to infer information.

In our joint work SPOILER, Saad Islam, Daniel Moghimi, Berk Gulmezoglu and Berk Sunar discovered timing differences during memory address resolution. Similarities between physical addresses could thus be spotted despite memory virtualization, allowing us to improve several existing attacks. Moritz Krebbel, Thomas Eisenbarth and I used the newly found leakage to improve eviction set construction. We showed that it can even be used from the browser, despite JavaScript adding an extra layer of virtual address abstraction. JavaScript can be used in *drive-by attacks*: No software installation, native code execution or special user privileges are required.

Parts of Section 4.3 are taken directly from the publication. We went through a responsible disclosure process and received a CVE[1]. I developed the algorithm for eviction set construction, wrote the corresponding part of the USENIX publication that introduced SPOILER [105], and took part in writing and editing the remaining paper.

As described in Subsection 2.6.2, processes work in their own virtual address space for various reasons. One of them is the security aspect of process isolation: Processes should not know the physical addresses of their own data so they cannot infer other processes' data locations. Information about the physical address space

[1] `https://nvd.nist.gov/vuln/detail/CVE-2019-0162`

is restricted to user processes. Consequently, behavior that leaks information about the physical address space poses a security risk. We show that the additional address information can be used directly to improve cache attacks as well as Rowhammer attacks and can be used as a covert channel. Optimizing attacks is important research as defenses can only be effective if the attacks have a realistic range. A countermeasure developed on a sub-optimal attack may clearly not defend against an optimized version.

4.1 The Intel Address Resolution

The processor has a Memory Order Buffer (MOB) to efficiently manage memory operations. As the processor often uses data from the cache, it needs to be tightly connected to the MOB. According to the Intel memory ordering rule, `store` operations are executed in-order, while `load` operations can be executed out-of-order [146]. A `store` will be added to the *store buffer* before being committed to memory. Both this and the out-of-order execution of `load` instruction allow the processor to continue executing while a comparatively slow commitment of the `store` is still pending.

To avoid working on stale data, a `load` instruction triggers an address comparison in the store buffer. The implementation of Intel's store buffer is undocumented. Reverse engineering efforts suggest that it holds the virtual address only, but may include part of the physical address in the comparison [6, 7, 124]. As the physical address is not compared before the `load` request is answered from the store buffer, data may be forwarded falsely. Before the `load` can be committed, the TLB needs to be involved for a complete address resolution, which is time consuming. The `load` is thus executed *speculatively* and, if a conflict is detected later on, will be rolled back and re-issued. The rollback results in a measurable timing difference.

Taking a closer look at the dependency resolution algorithm, we could only find various suggestions in Intel patterns, but no thorough documentation. A potential design was described by other authors before, and is in agreement with our observations [90, 124]. Dependencies are checked in three steps, which are shown in Figure 4.1: A *loosenet step*, a *finenet step* and the complete physical address resolution.

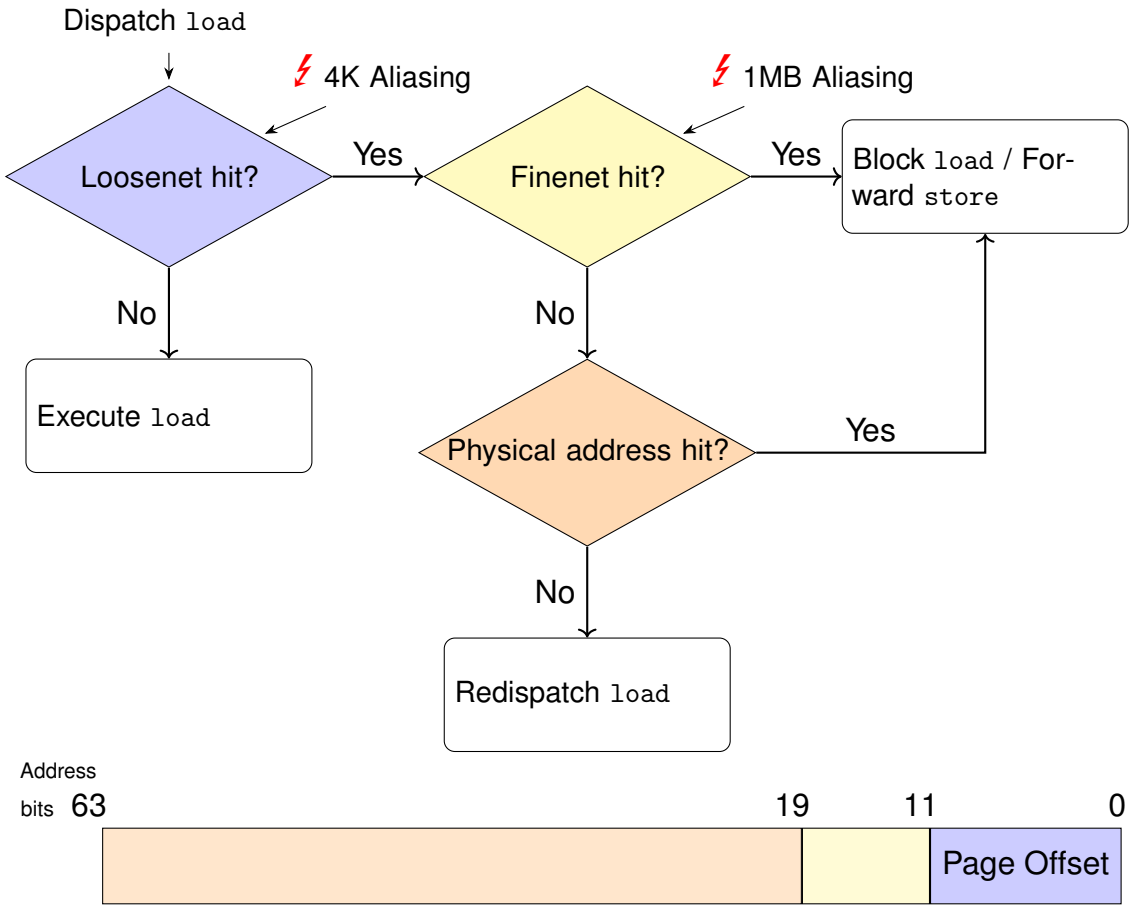

Figure 4.1: The dependency check logic: *loosenet* initially checks the 12 least significant bits (page offset). Depending on the implementation, the *finenet* checks the upper virtual address bits (12 to 19), or the physical address tag. The final dependency using the physical address matching might still fail due to partial physical address checks, in which case the load is re-dispatched to give the address resolution more time. This figure is drawn after Figure 3 of [105]. The lightning bolts show places in the dependency resolution that suffer from false positives due to aliasing effects.

The loosenet step compares the page offset of the `load` and the `store`. As the page offset is the same in the virtual and physical address, no address translation is required. If there is no hit, the `load` is independent from the `store` and can proceed. In case of a loosenet hit, the *enhanced loosenet* or *finenet* check is next. There are different suggestions as to how it is implemented, either to check the upper virtual address bits [90], or the physical address tag [124]. If a finenet hit is detected, the load is blocked and the store is forwarded. However, it is not the final stage

of the dependency resolution. When there is no hit, the physical address will be compared. Some publications such as [7] suggest that the store buffer only holds bits 19 to 12 of the actual physical address. In any case, the finenet check may return false dependencies, which indicates that the comparison does not use all physical address bits, most likely because they are not available in the memory order buffer despite being present in the physical address buffer.

4.2 Measuring the SPOILER Effect

The SPOILER effect was measured and then confirmed by various Hardware Performance Counters (HPCs). HPCs allow developers to monitor low-level hardware events within the CPU, providing valuable insights into program performance. By accessing special purpose registers, developers can analyze and identify microarchitectural bottlenecks, while libraries like PAPI simplify the process of collecting and interpreting HPC data on Intel processors. The HPCs used to examine the SPOILER effect are described in Table 4.1.

Table 4.1: The hardware performance counters used to measure and categorize the SPOILER effect. Other counters were evaluated, but did not show a relevant correlation.

Counters	*Correlation*	Description
`Cycle_Activity:Stalls_Ldm_Pending`	0.9819	Only counts memory accesses whose latencies could not be avoided by out-of-order execution [161]
`Ld_Blocks_Partial:Address_Alias`	-0.9511	Counts false dependencies in MOB when the partial comparison upon loose net check and dependency was resolved by the Enhanced Loose net (finenet) mechanism. Loose net checks can fail when loads and stores are 4k aliased [97].
`Exe_Activity:Bound_on_Stores`	0	Counts cycles where the Store Buffer was full and no `load`s caused an execution stall [97].

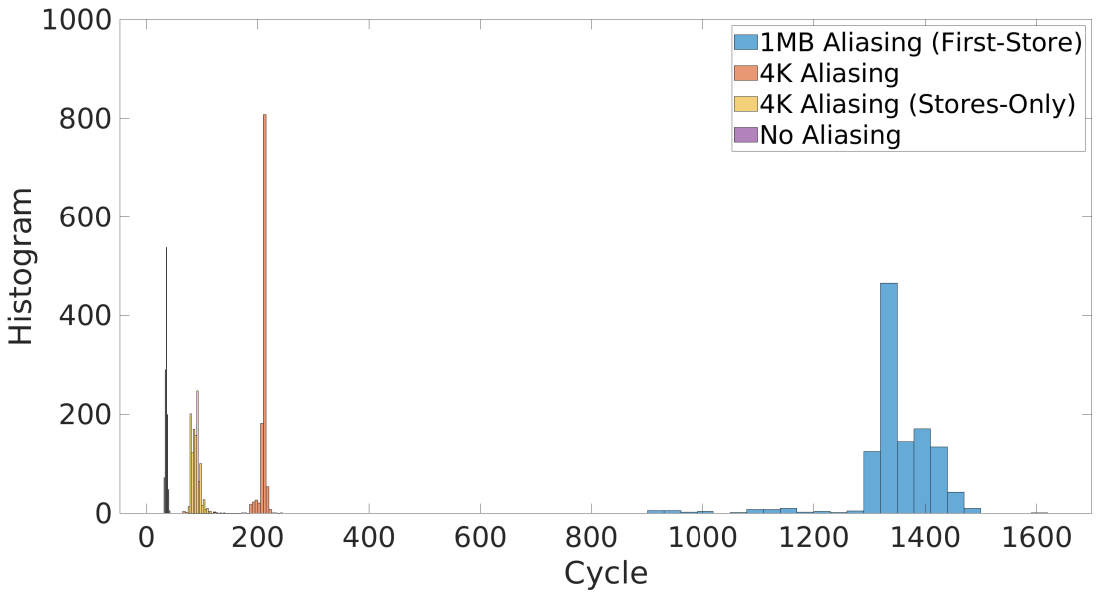

Figure 4.2: Histogram of the measurement for the speculative load with various store addresses. `Load` will be fast, 30 cycles, without any dependency. If there exists 4K aliasing only between the `stores`, the average is 100. The average is 200 when there is 4K aliasing of `load` and `stores`. The 1MB aliasing has a distinctive high latency. Figure from [105].

The event `Cycle_Activity:Stalls_Ldm_Pending` had the highest correlation of 0.985. This event shows the number of cycles for which the execution is stalled and no instructions are executed due to a pending `load`. We can conclude that the SPOILER effect is caused by a blockage in the load pipeline. The event `Ld_Blocks_Partial:Address_Alias` has an inverse correlation with the leakage. This event counts the number of false dependencies in the MOB when the loosenet check resolves the 4K aliasing condition: The lower 12 bits of the address matched, but the next compared bits do not. The counter `Exe_Activity:Bound_on_Stores` increases with more number of `stores`, but it does not have a correlation with the leakage: Even though the store buffer is full, the timing behavior is not due to the `stores`.

Figure 4.2 shows the latencies introduced by different aliasing effects in address resolution. They are marked with red arrows in Figure 4.1.

Result: False dependencies in the store buffer generate a high latency in executing load instructions.

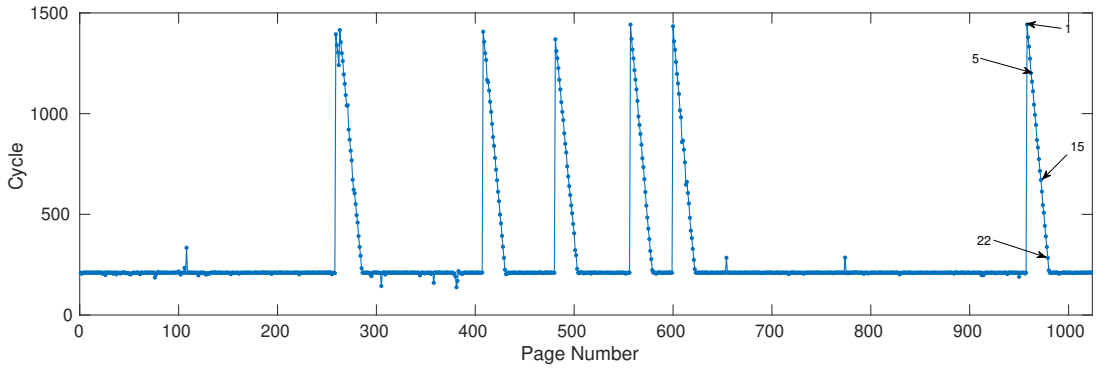

Figure 4.3: Step-wise peaks with 22 steps and a high latency can be observed on some of the pages (*Core i7-8650U* processor). Figure from [105].

To determine how much address information can be gained from the speculative load effect, the physical page numbers of addresses showing the latency are observed in the `pagemap` file. Note that this file is only available to users with elevated privileges. As shown in Figure 4.3, step-wise peaks with a very high latency appear once in every 256 pages on average. The 20 least significant bits of physical address for the `load` matches with the physical addresses of the `stores` where high peaks for virtual pages are observed. In our experiments, we always detect peaks with different virtual addresses, which have the same 20 least significant bits of the physical address. This observation clearly indicates the existence of a 1 MB aliasing effect based on the physical addresses. We thus discovered a 1 MB aliasing effect which leaks information about 8 bits of mapping that were unknown to the user space processes.

When matching this observation with the previously discussed performance counter `Cycle_Activity:Stalls_Ldm_Pending` with a high correlation, we can conclude that the speculative load has been stalled 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, `Ld_Blocks_Partial:Address_Alias` drops to zero, and it increments at each down step of the peak. We can conclude 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.

Result: The latency can be observed if 20 bits of the physical address overlap, thus leaking 8 bits of physical address information.

The address resolution and the conducted experiments are described in more detail in [105].

4.3 Micorarchitectural Attacks from JavaScript

Microarchitectural attacks from JavaScript have a high impact as drive-by attacks in the browser can be accomplished without any privilege or physical proximity. In such attacks, co-location is automatically granted by the fact that the browser loads a website with malicious embedded JavaScript code. The browsers provide a sandbox where some instructions like `clflush` and `prefetch` and file systems such as `procfs` are inaccessible, limiting the opportunity for attack. Genkin et al. showed that side-channel attacks inside a browser can be performed more efficiently and with greater portability through the use of *WebAssembly*. WebAssembly is a bytecode language intended to provide a portable compilation target for developers. While it was intended to increase performance, WebAssembly poses a security risk. It works with unmanaged memory, opening security risks long managed in native binary code: Lehmann et al. showed that WebAssembly code is susceptible to Buffer overflows, stack overflows, and randomly overwriting memory [128] Despite being a sandboxed environment, it aided Spectre attacks from web browsers, and is often used in crypto mining [143]. In 2019, Musch et al. analyzed the Alexa 1 Million web pages and found that over 50% of the pages using web assembly use it in a malicious way [165].

WebAssembly introduces an additional abstraction layer, i.e. it emulates a 32-bit environment that translates the internal addresses to virtual addresses of the host process (the browser). WebAssembly only uses addresses of the emulated environment and similar to JavaScript, it does not have direct access to the virtual addresses. Using SPOILER from JavaScript opens the opportunity to puncture these abstraction layers and to obtain physical address information directly. Figure 4.4 shows the address search in JavaScript using SPOILER. Compared to native implementations, we replace the `rdtscp` measurement with a timer based on a shared array buffer [84]. We cannot use any fence instruction such as `lfence`, and

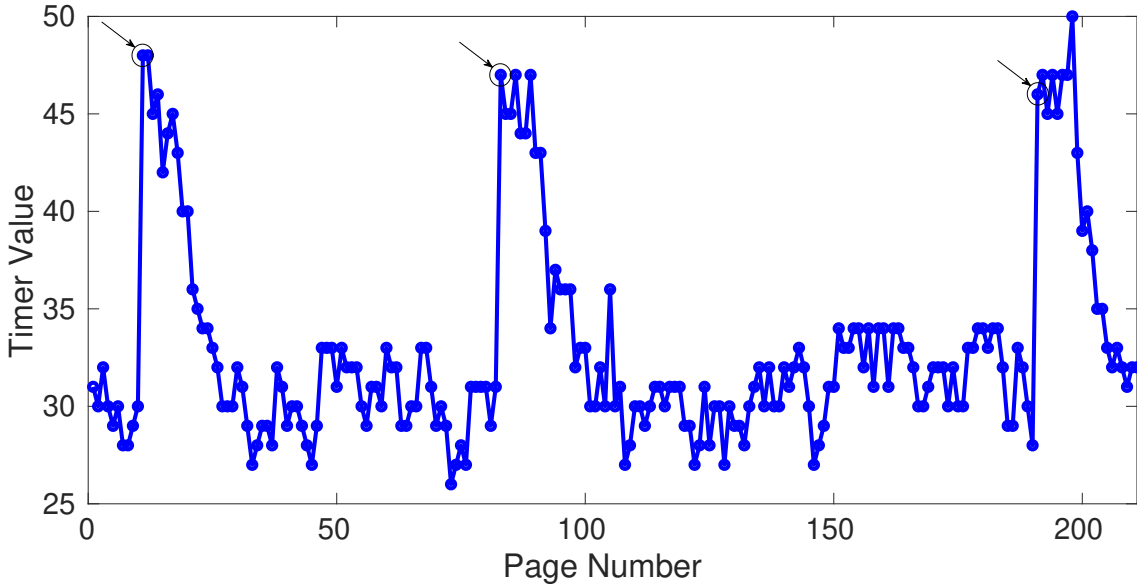

Figure 4.4: The SPOILER leakage in JavaScript. It can be used to reverse engineer physical addresses. The markers point to addresses having the same 20 least significant bits of physical addresses. Figure from [105].

as a result, there remains some negligible noise in the JavaScript implementation. However, the aliased addresses can still be clearly seen, and we can use this information to improve the state-of-the-art eviction set creation for both Rowhammer and cache attacks.

4.3.1 Improved Eviction Set Finding

As mentioned in Subsection 2.9.1, many cache attacks require so-called *eviction sets*. An eviction set is a set of memory addresses that completely covers a cache set. Usually, the set should be minimal, meaning an eviction set for a cache with w ways has w elements. For other uses cases such as cache fingerprinting of websites, not only one cache set needs an eviction set, but all of them [173]. The easiest way of creating eviction sets is by using *huge pages*: This CPU-feature allows for the allocation of very large memory pages that have a starting address that is a multiple of 2MB in both physical and virtual address space. As the allocated memory is consecutive, the process allocating the huge pages knows the least significant 21 bits of the addresses, which makes it trivial to find an eviction set [139]. There are, however, situations where the use of huge pages is not an option: A

cloud vendor could disable the feature, or additional layers of abstraction such as in portable code do not support the feature. In that case, eviction sets need to be constructed with limited address information.

Oren et al. constructed eviction sets from JavaScript by allocating a large amount of virtual memory [173]. The virtual memory is designated to the process by the browser and is not continuous. As virtual addresses are page aligned on Intel processors, the last twelve bits of the virtual address are also the last twelve bits of the physical address. Six of these bits are used to determine the cache set (see Section 2.9 for details). Using this information about physical addresses, the initial algorithm suggested in [173] uses a randomized algorithm to brute-force eviction sets from a set of pages. Genkin et al. refined this algorithm by removing some of the randomness. They construct the eviction set in three stages: *expand*, *contract* and *collect*. The expand-step is depicted in Figure 4.5: The algorithm starts with an address pool of page-aligned addresses and picks a *witness address* x as well as w addresses as the initial eviction set S. Addresses from the address pool are subsequently added to S. After adding each address, x is loaded, then all addresses of S, and then x again. If the loading time of x suggests a cache miss, S is an eviction set for x and the expand step is finished. As the mapping is unknown, many addresses that map to other cache sets than x will also be part of S.

During the *contract* phase depicted in Figure 4.6, addresses are removed from S one by one. After removing each address, the algorithm verifies whether S still evicts x. If it does, the algorithm continues to remove the next address from S. If it does not, the address was a part of the minimal eviction set for x and is re-added to S before continuing with the next address.

In the last phase of the algorithm, the collect phase, all known six bits of the page aligned addresses in S are enumerated to gain 63 additional eviction sets.

We improved the process of finding eviction sets in several steps. First, we added all addresses that got discarded from S_i to S_{i+1} during the *Contract*-phase: As an address that is not part of eviction set S_i has to be part of a different set, it is a good idea to start the next set with the discarded addresses from the previous run. The contrast at the beginning of the expand phase for set S_{i+1} is illustrated in Figure 4.7. It shows that the likelihood of having correct addresses in the eviction set from the beginning of the expand phase increases. As the address pool gets smaller and smaller, this advantage gets more prominent. We tested this approach on an Intel Core i7-4770 with four physical cores and a shared 8MB 16-way L3 cache

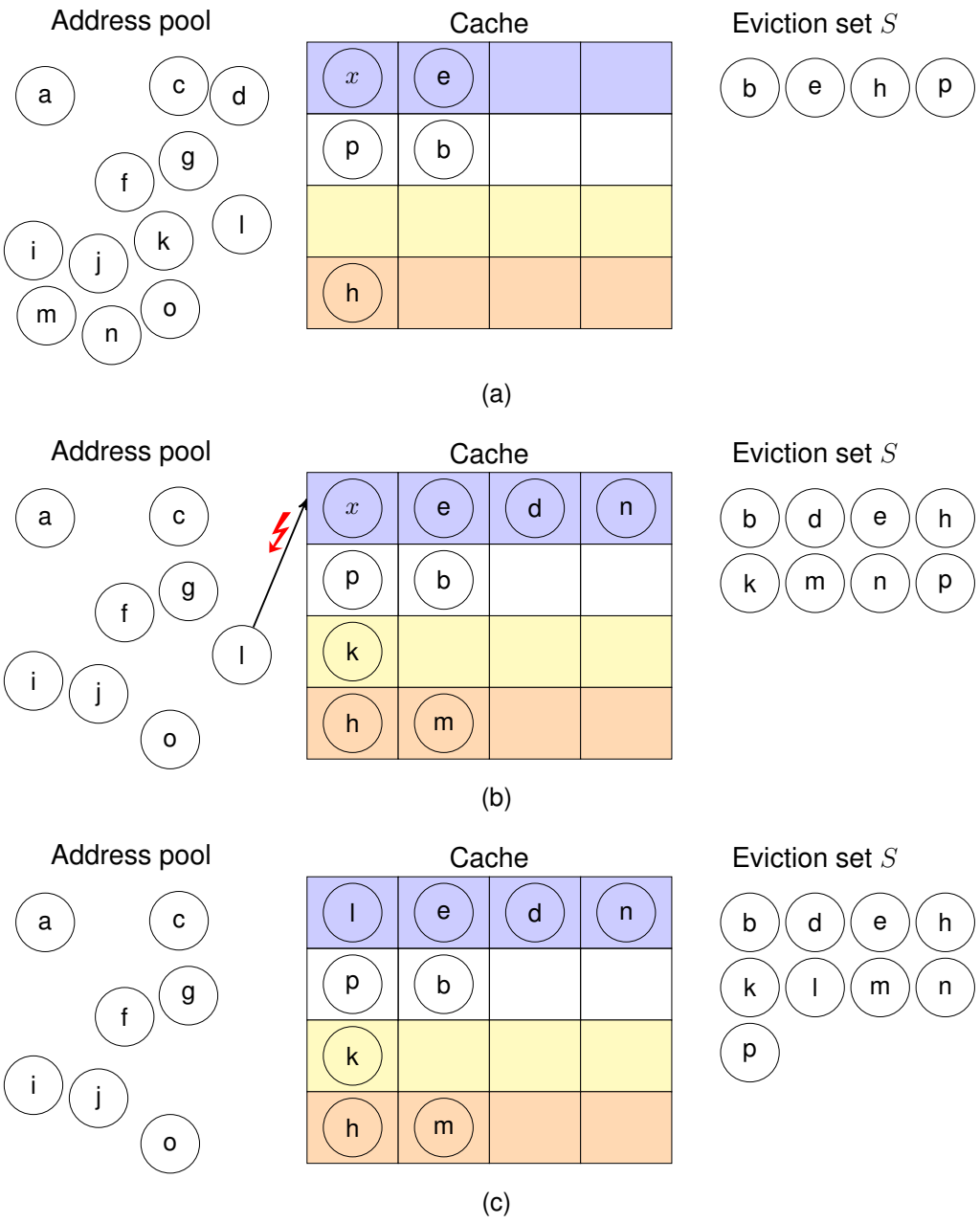

Figure 4.5: The expand phase of the eviction set finding algorithm for a 4-way cache. The address pool consists of page-aligned addresses. (a) The witness address x is loaded into the cache, the eviction set S contains w randomly picked addresses. (b) Random addresses from the address pool are added until the cache set the victim x maps to is full, creating a collision with x (c) The expand phase is finished once x got evicted.

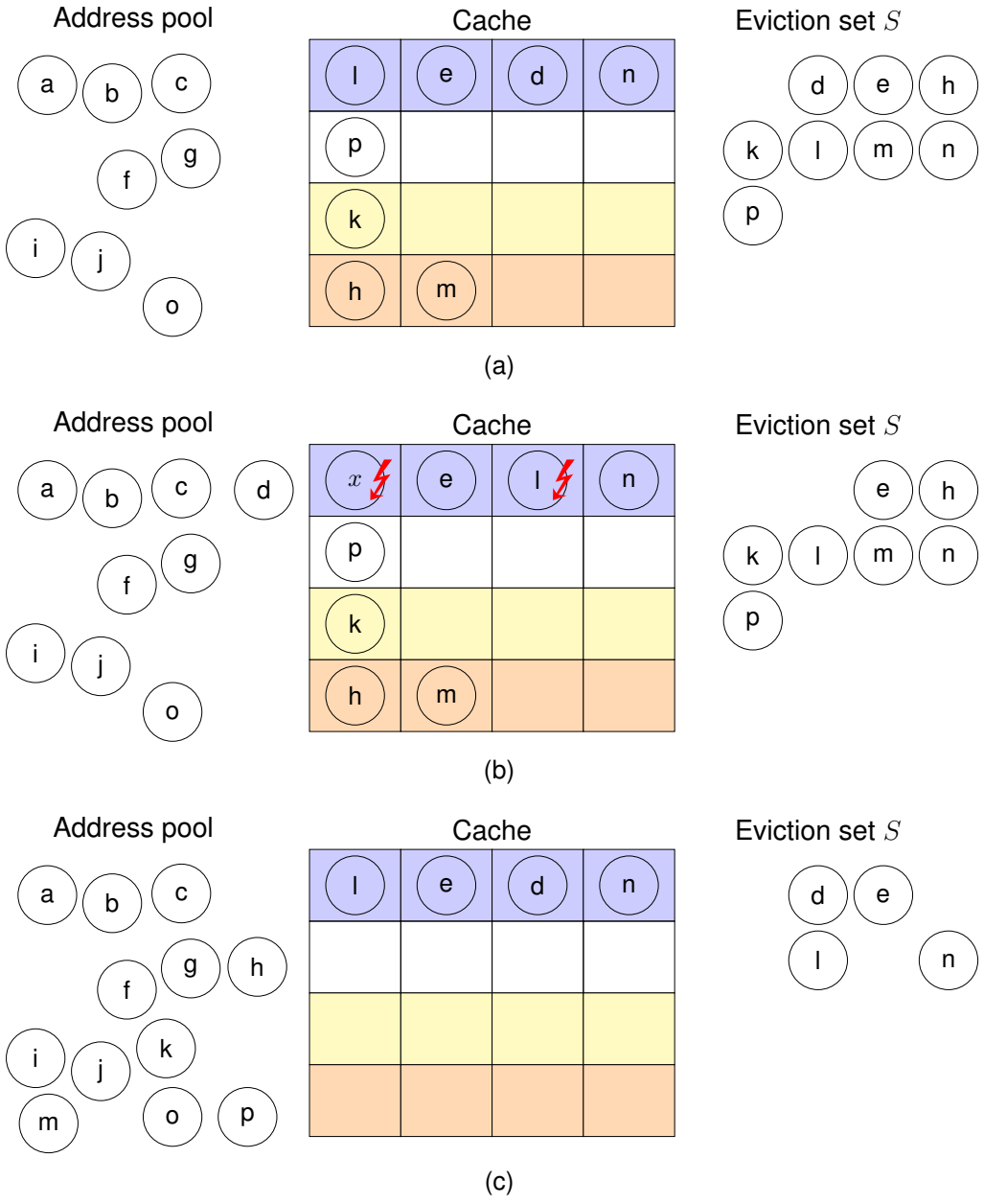

Figure 4.6: The contract phase of the eviction set finding algorithm for a 4-way cache. The address pool consists of page-aligned addresses. (a) b is removed from the eviction set S. As S still evicts x, b is not needed and re-added to the address pool. (b) As address d is removed from the address pool, x is not evicted anymore as the former colliding address l can now be mapped to the empty cache line. d is thus re-added to S. (c) The contract phase is finished once $|S|$ is exactly w.

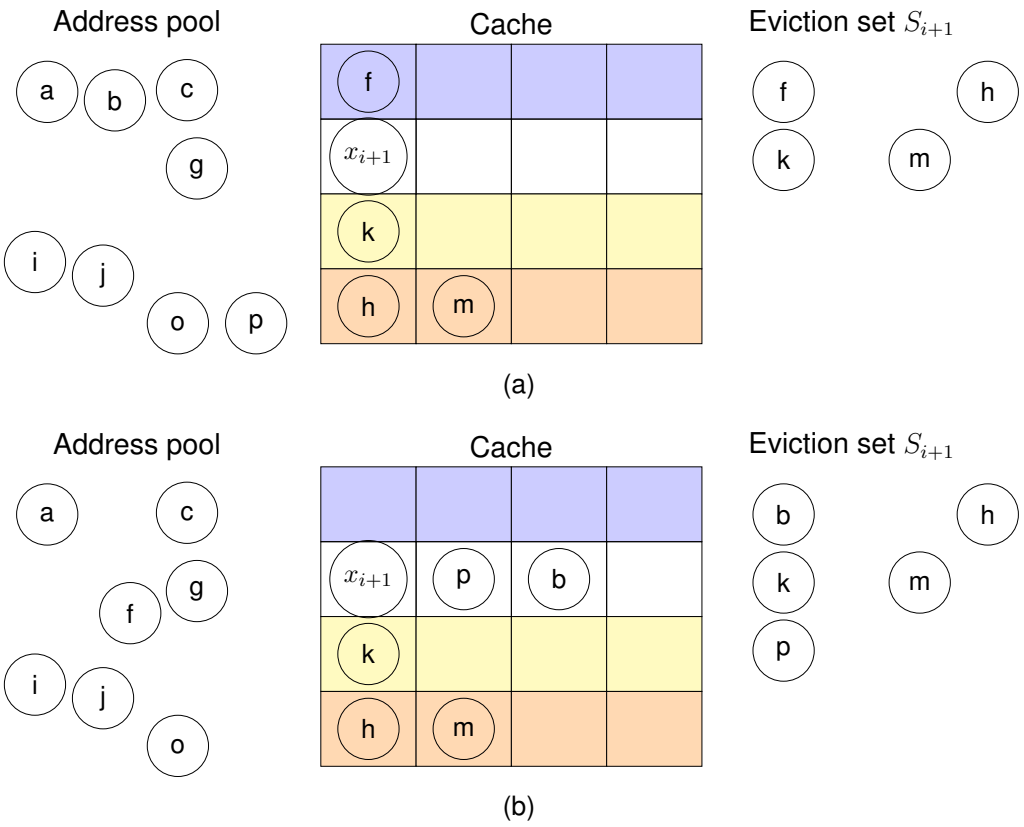

Figure 4.7: New expand phase: (a) In the old algorithm, a new witness address x_{i+1} is picked and four random elements are added to S_{i+1}. (b) The addresses discarded from the eviction set S_i for x_i during the contract phase are directly added to the eviction set S_{i+1} for the next witness x_{i+1}.

with Chromium 68.0.3440.106, Firefox 62 and Firefox Developer Edition 63. The approach yields an 80% accuracy rate to find all 8192 eviction sets when starting with a pool of 4096 pages. The entire eviction set creation process takes an average of $46\,s$ with the original algorithm proposed by Genkin et al. The improved eviction set creation process takes $35\,s$ on average.

However, the SPOILER effect helped us to decrease the runtime for finding eviction sets further. In the original address pool, the probability of finding an address that maps to the same eviction set as the witness x is $P(C) = 2^{\gamma-c-s}$, where c is the number of bits determining the cache set, γ is the number of bits the attacker knows, and s is the number of slices [221]. The additional information provided by SPOILER sets $\gamma \geq c$, which only leaves the few address bits used by the slice

selection algorithm as uncertainty [104]. The address pool needs to consist of aliased addresses. Generating such a pool takes about 1 second locally, but due to noise in JavaScript, it takes 9 seconds for finding aliased addresses. Finding all eviction sets then takes an additional 3 seconds. The success rate is 100% with SPOILER as compared to 80% for the classic method. The different algorithms and improvements are summarized in Table 4.2

Result: Using aliased address pools improves the time required to completely cover the cache with eviction sets from 46 seconds to 12 seconds while improving the success rate to 100%.

Table 4.2: Comparison of different eviction set finding algorithms on an Intel Core i7-4770. *Classic* is the method from [173], *Pre-fill* is the same method, but pre-fills the next eviction set with the discarded values from the previous one, *Aliased Address* uses SPOILER. t_{AAS} is the time percentage used for finding aliased addresses. t_{ESS} is the time percentage for finding eviction sets. R is the number of Rounds.

Algorithm	R	t_{total}	t_{AAS}	t_{ESS}	Success
Classic [173]	3	46s	-	100%	80%
Improved [68]	3	35s	-	100%	80%
Aliased Addresses (ours)	10	10s	54%	46%	67%
Aliased Addresses (ours)	20	12s	75%	25%	100%

As there are 4 slices and they are the only unknown part of the eviction set mapping, each aliased address pool contains addresses from 4 eviction sets. These can be enumerated again to form 63 more eviction sets since we still kept the bits 6-11 fixed. To accomplish full cache coverage, the aliased address pool has to be constructed 32 times. As SPOILER increases both speed and reliability of the eviction set finding, it can be used to improve an end-to-end attack such as drive-by key-extraction cache attacks by Genkin et al. [68].

4.4 Further Uses of SPOILER

In the following, we look at further uses for SPOILER: It can be used to improve the accuracy of Rowhammer attacks or as a covert channel to convey information

across context switches. We also tried to use the SPOILER effect to channel secrets out of a TEE, which proved to be impossible.

4.4.1 Rowhammer

In our publication [105], SPOILER was additionally used to improve Rowhammer style attacks. Rowhammer is an interesting example of a hardware effect from a software access, and, in turn, software malfunction due to a hardware vulnerability. It was first described in 2014, when Kim et al. found that they could induce faults to memory addresses by repeatedly accessing ("hammering") a physically adjacent address line. Fault attacks can be very powerful. It is, for example, enough to flip a single bit in an RSA signature to obtain the private key [117]. Fault attacks can also be used to elevate privileges, or to simply attack the integrity of the victim's data by changing them. The physical address information gained from SPOILER helps to increase the probability of bank co-location, allowing us to run the first double-sided Rowhammer attack with user privileges.

Result: The SPOILER effect can be used to find addresses co-located on a bank, which improves Rowhammer attacks.

As the Rowhammer attack needs to access DRAM and thus bypass the cache, it requires the possibility to remove data from the cache. In absence of a `clflush` instruction, this can be done via an eviction set. In that case SPOILER is doubly useful: It can be used to find the continous memory for the Rowhammer attack and to speed up the eviction set generation.

4.4.2 Covert Channel

Covert channels are used to transform information from one security context to another [121, 136, 156]. If an attacker process runs before/after a victim process on the same thread, parts of the microarchitectural state are visible after the context switch. The switch may be between user processes, user space and kernel space, or even into or out of a TEE [156, 217]. The microarchitectural state thus transports information out of the process that changed it, making e.g. the cache a covert channel.

We examined whether the SPOILER effect can be used as a covert channel, i.e. whether the store buffer is emptied upon context switches. We filled the store buffer with arbitrary addresses, issued a context switch to a process performing a secret-dependent memory access, and measure the execution time of the victim process. Any correlation between the victim's timing and the load address can leak secrets [237]. The attack needs to be timed to ensure the victim accesses the memory while there are aliased addresses in the store buffer, otherwise there will not be a speculative load and thus no resolution hazard. We find that we can, indeed, observe the victim access after the context switch.

To further quantify the covert channel, we perform an analysis of the number of non-memory dependent operations that can be executed between the `stores` and the `load`. We use the `add` instruction as it typically takes about 1 cycle. We found that about 1000 `adds` can be executed between the `stores` and `load` before the SPOILER effect is lost.

We try to track a secret dependent memory access in the privileged kernel environment. The attack has three steps:

1. Fill the store buffer with addresses that have the same page offset.

2. Execute a system call.

3. Measure the execution time.

If the secret memory access aliases with the addresses in the store buffer, we will see a delay. We use the syscall `mincore`, a kernel function that checks whether a certain address is in RAM or needs to be fetched from the disk[2].

As displayed in Figure 4.8, we can see a delay with 7 steps, allowing us to track the address of the kernel memory `load` if our arbitrarily picked addresses alias with the `load`. The blue line (No conflict) 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 is likely due to many addresses having to be checked for conflicts if the store buffer is full.

[2] `https://man7.org/linux/man-pages/man2/mincore.2.html`

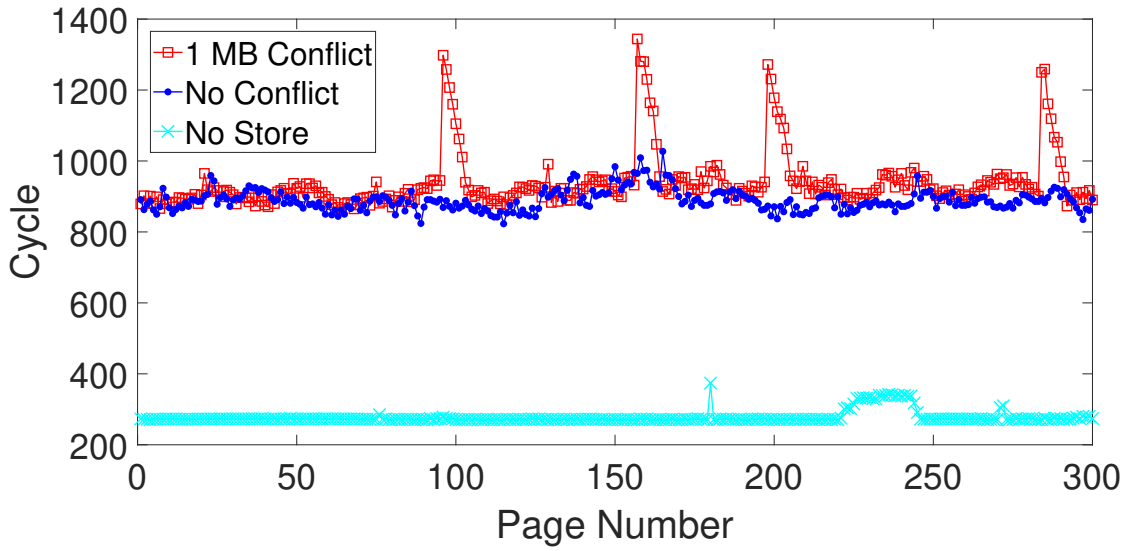

Figure 4.8: Execution time of `mincore` system call. When a kernel `load` address has aliasing with the attacker's `stores` (red/1MB Conflict), the step-wise delay will appear. These timings are measured with Kernel Page Table Isolation disabled. Figure from [105].

4.4.3 No Leakage from SGX

We tried to combine the CACHEZOOM attack [156] with SPOILER to infer secrets that were accessed within an SGX enclave. We use *SGX-STEP* by Bulck et al. to precisely interrupt every single instruction [30]. The same group showed that the interrupt handler context switch time is dependent on the execution time of the currently running instruction [217]. As a baseline, we measure a context switch on our test platform. It takes about 12 000 cycles to execute. We then continue as in the previous experiment, filling the store buffer with addresses matching the page offset of a `load` inside the enclave and measuring whether we can see a difference in timing depending on the loaded address. While the time required for the context switch increases to 13 500 cycles, we cannot observe any correlation between matched addresses. We do, however, see stepwise *increases* in performance that look like the SPOILER leakage without any correlation. Later, we observe a similar pattern by running SPOILER before an `ioctl` routine that flushes the TLB at each call. Intel SGX also performs an implicit TLB flush during each context switch. We can thus infer that the downward peaks occur due to the TLB flush, especially since the addresses for the downward peaks do not have any address correlation with the `load` address. We did not investigate this behavior further and just concluded

that the TLB flush seems to be effected by SPOILER, which makes it impossible to infer any information from inside the enclave.

Responsible Disclosure

We informed the *Intel Product Security Incident Response Team* (iPSIRT) of our findings. iPSIRT thanked us for reporting the issue and for the coordinated disclosure. iPSIRT then released the public advisory and CVE. The timeline for the responsibel disclosure was as follows:

- **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-2019-0162.

4.5 Mitigation and Countermeasures

SPOILER exploits the fact that when a `load` instruction is issued after a number of `store` instructions, the addresses in the store buffer are checked for conflicts with only partial address information. The partial address check can lead to false dependencies, which cause severe timing behavior during resolution. The root cause is the speculatively executed `load` before all the `stores` are finished executing.

There is *no software mitigation* that can completely erase this problem. The timing behavior can be removed by inserting store fences between the `loads` and `stores`, but this cannot be enforced in the user's code space. Intel is thus not able to ensure that processes do not leak physical address information, but developers could avoid the timing behavior by introducing fences. Since many developers are not familiar with side channel attacks and since fences reduce performance, it is unlikely that these countermeasures will be widely adopted. Executing other instructions between the `loads` and the `stores` to decrease the depth of the attack is even less effective, as the compiler may optimize these away. As always, the performance penalty introduced by these countermeasures can severely lower their acceptance. As for most attacks on JavaScript, removing accurate timers from

the browser would be effective against SPOILER. Indeed, some timers have been removed or distorted by jitters as a response to attacks [136]. Despite the efforts to make timing attacks harder, there is a wide range of timers with varying precision available, and removing all of them seems impractical [65, 194]. Recently, Katzman et al. managed to store information on the cache state of a victim at a different location in the cache, making sampling rates independent of measuring rates. They manage to slow down their measurements so much they can gain information while working with a timer accuracy of 0.1 microseconds.

Since at this point it does not seem realistic to mitigate the microarchitectural leakage from a software perspective, developers can use dynamic tools to at least *detect* the presence of such leakage [23, 43, 241]. For example, they can monitor hardware performance counters in real-time. As explained in Section 4.2, we found that particularly the two performance counters `Ld_Blocks_Partial:Address_Alias` and `Cycle_Activity:Stalls_Ldm_Pending` show high correlations with the leakage. Monitoring these can inform an administrator that an attack may be going on, and take appropriate measures such as switching off the process, switching to a different server or trying to find out which process is performing the attack.

The *hardware design* for the memory disambiguation may be revised to prevent such physical address leakage, for example by avoiding speculative behavior or implementing a full address resolution and comparison straight after the loosenet check. Of course, resolving and comparing a larger part of the address will have performance impacts. It is safe to assume that partial address comparison was a design choice for performance. Moreover, hardware patches are difficult to be applied to legacy systems and take years to be deployed. At the time of writing of this thesis, the timing behavior is still present.

4.6 Related Work

SPOILER exploits the speculative behavior during address resolution of Intel's proprietary implementation of the memory subsystem, which directly leaks timing behavior due to physical address conflicts. A previously studied false dependency resulting in timing behavior is the 4K aliasing false dependency on Intel processors [62, 237], which is documented by Intel [100]. *MemJam* [155] uses this behavior to perform a side-channel attack, and Sullivan et al. [206] demonstrate a covert channel. The

authors conclude that the 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. On the contrary, we discover that the undocumented *address resolution* logic performs additional partial address checks that lead to observable aliasing behavior based on the physical address.

Several microarchitectural attacks have been discovered to recover virtual address information by exploiting the Translation Look-aside Buffer (TLB) [93], Branch Target Buffer (BTB) [61] and Transactional Synchronization Extensions (TSX) [108] as well as timing information obtained from the `prefetch` instruction [77]. The main obstacle to this approach is that the `prefetch` instruction is not accessible in JavaScript, and it can be disabled in native sandboxed environments [238]. SPOILER is applicable in both cases. Knowledge of additional address information facilitates microarchitectural attacks, such as cache attacks on ARM processors [135].

Browsers are an attractive platform for attacks as the victim only has to open a website to be vulnerable. Co-location with the attacker is given. It is often enough for the attacking website to be open in a different tab. JavaScript has thus been used in various attacks: Shusterman et al. mount a JavaScript cache attack that uses ML-techniques to fingerprint websites opened in other tabs [200]. In the same year, Shusterman, Genkin and Oren (among others) start with a JavaScript attack and develop it away from specific scripting features until they have a pure HTML and CSS side channel attack that is independent of hardware [199]. The browser can be used to induce faults on the victim machine [78]. Speculative behavior can leak secrets via JavaScript [121] or even allow the attacker to read arbitrary memory [195]. In 2023, Katzman et al. used transient execution attacks to build turing-complete gates in the cache of a victim via the browser. They also used their technique to create eviction sets, and do so with 83% success rate in $15s$. Their algorithm's success rate is slightly worse than our algorithm's, however the algorithm by Katzman et al. is platform independent and can thus not leverage the SPOILER leakage. Recently, Giner et al. constructed a drive-by attack on a GPU and managed to infer keystrokes as well as recover a key from an AES T-Table implementation via GPU eviction sets.

4.7 Summary and Conclusion

We discovered SPOILER, a previously unknown 1MB aliasing effect in the Intel address resolution. When the store buffer is full, loads will be executed speculatively since the store buffer is unable to truly resolve dependencies with the available address information. Once the false dependency is discovered, it leads to a load delay that can be measured by an observer. The aliased addresses share the same 20 least significant bits of their *physical* address, thus leaking 8 bits of physical address information to an observer. We can see a lasting impact on the research community when looking at the follow up works for SPOILER.

The possible causes of the leakage were analyzed in detail by monitoring performance counters and observing the leakage under various circumstances. Additionally, we studied Intel patterns and documentation. We concluded it stems from the *finenet* check in the Intel address resolution hierarchy, which is undocumented. While the previous loosenet check works on the page offset, the finenet check seems to include 8 more physical address bits into the dependency resolution, but not the entire address. We thus gained deeper insight into previously unknown behavior of Intel processors, which is useful to further research and secure systems. We then demonstrated the usefulness of this new information in several ways.

Eviction set finding Firstly, we used it to develop an algorithm to construct eviction sets almost 4 times more efficiently than the previous state of the art. the algorithm is applicable from JavaScript, which means physical address information becomes available in an extremely restricted, sandboxed environment. Efficient eviction set finding is a relevant topic until this day [182, 210]. The algorithm by Vila et al. is often used as the baseline [114, 182, 210], although Song and Liu presented an improved version and a thorough analysis at the same conference we presented SPOILER. Finding out how well their algorithm translates to a portable code variant or whether SPOILER can improve their variant even further is a good direction for future work. We only compare to other portable code implementations. Katzman et al. developed a platform independent approach that takes $15s$ and has 83% accuracy. Kim et al. attack apple devices from the browser and take longer than the baseline, but have a high probability of success even with vastly degraded timers or in timerless environments [116]. Purnal et al. also try to tackle the problem of missing or low resolution timers. They find eviction sets with no error rate, but

require time frames between milliseconds and days [182]. Briongos et al. used the SPOILER leakage to create eviction sets outside of JavaScript [24].

Rowhammer Secondly, we used the information to perform a highly targeted Rowhammer attack in a native user-level environment. Rowhammer attacks are highly relevant up to this day [109, 123, 163, 164, 218, 219], and the SPOILER leakage is still used in our institute to ease these attacks during research. We show the first double-sided Rowhammer attack without elevated privileges. As we provide a JavaScript implementation of SPOILER, it can easily be integrated into the existing JavaScript version of Rowhammer, removing the assumption of 2MB huge pages [78]. Many researchers use SPOILER to find continuous memory since they assume an unprivileged attacker [9, 106, 163, 164, 211].

Covert channel The leakage can also be used as a covert channel to transfer information over context switches. Chakraborty et al. build a *Fill-and-Missdirect* attack around the leakage and use it to extract AES keys from a T-Table implementation [35]. SPOILER is not applicable to infer secrets from Intel SGX.

Canella et al. found a new application for the SPOILER effect. They use it to slow down their process to improve their measurements [33]. Additionally, SPOILER was mentioned in various systematization of knowledge papers [32, 87, 240]. We can thus state that it has a lasting impact on the security research community.

5

CacheSniper: Aiming for Weaknesses in Side Channel Defenses

In the previous chapter, we discussed several examples of microarchitectural side channel attacks. The SPOILER leakage is a side channel for address information to unprivileged processes. We also discussed how the additional information can be used to build eviction sets, a fundamental building block of many cache side channel attacks. This chapter will be dedicated to the analysis of potential defense mechanisms against these attacks: Prefetching and always-load strategies.

In our joint work CACHESNIPER, Samira Briongos and me thoroughly analyzed the conditions under which an attacker can use very small windows to attack cryptographic implementations. By carefully controlling the cache state in combination with a reverse engineered cache policy, a prefetch protected implementation is still vulnerable. Samira Briongos, Pedro Malagón and José M. Moya used the extreme control over the cache state to break a side channel defense in an AES implementation. Thomas Eisenbarth and me applied the attack to an RSA implementation. I found a vulnerable RSA implementation, also with side channel protections already in place, from which we could extract the key. I substantially contributed to the writing of the publication and coordinated the disclosure process. While the vulnerability of the AES implementation was considered out of scope for the attacked library OpenSSL, we went through a responsible disclosure process for the RSA vulnerability in WolfSSL and received a CVE[1]. We helped WolfSSL to close the vulnerability. Parts of this chapter were also part of [22].

[1] `https://nvd.nist.gov/vuln/detail/CVE-2020-15309`

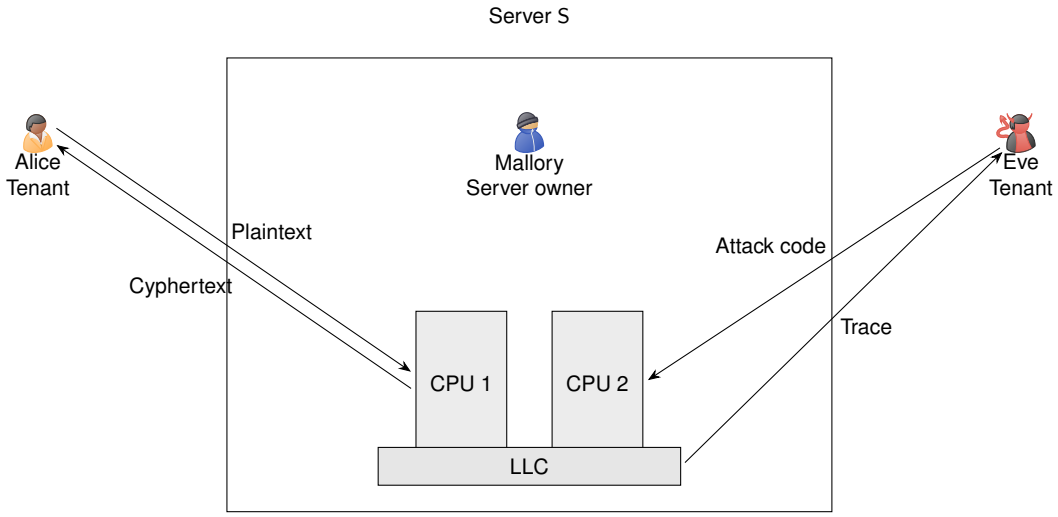

Figure 5.1: Adapted attack scenario for CACHESNIPER: To achieve the best results, the server S requires TSX. The attacked algorithm is cryptographic, AES and RSA respectively. The client Alice provides the plaintext, the attacker Eve obtains a cache trace that enables her to infer the cryptographic key. It is possible for the server owner to attack as well, it does, however, not need require this special role.

5.1 Attack Scenario

The scenario we consider to analyze of the effectiveness of the countermeasures and to carry out the different experiments is depicted in Figure 5.1. The server executes the target process, in our case an encryption/decryption with RSA or AES, upon request from an honest client. In our experimental scenario, the client sends requests to the server every $500\mu s$ plus a random time $\Delta \in [0..100\mu s]$. This process tries to emulate the behavior of a real network and enables us to show that we can actually detect a victim process execution. The attacker is also a tenant on the server and monitors the cache to detect the exact times when the target process (an encryption process) is running. She can later obtain a cache trace and analyze it to get cryptographic keys. Our main assumption is that the attacker and the victim are using the same machine. The attacker has user-level access to the server and no special rights or privileges. It is of course also possible to host the attack as server owner, but user level access is enough. The attacker does not know when the client starts executing and cannot physically interfere with the machine. As our attack targets the LLC, the attacker and victim may run on different physical cores.

5.2 Prefetch Protected Implementations

To secure algorithms against side channel attacks, implementations need to be constant in *time*, *execution flow* and *data usage* when using sensitive data. We illustrate with an algorithm using a secret key to encrypt input data: it should behave independent of input data and key. A possible implementation error regarding the time could be a direct multiplication of key bits with input bits: multiplication by zero or a power of two may be faster than a multiplication with other factors. Regarding execution flow, calling different functions or functions in different order depending on a key bit would be a common flaw [68]. Lastly, data usage is especially critical if key tables are precomputed: Usage of the precomputed values can be observed in the cache, directly leaking key bits to an attacker.

To achieve higher resistance to cache attacks, one countermeasure is easy to implement and often used: *prefetching*. Prefetching loads all data required for the algorithm into the cache ahead of usage if possible. It is supposed to ensure no cache access patterns can be observed as data and instructions are present in the cache regardless of their actual utilization. If the time between prefetch and actual access is low enough, it is not possible for an attacker to measure the cache state in between. That means an attacker cannot tell whether a piece of data present in the cache is there because of the prefetch or because of the subsequent use by the vulnerable calculation. Prefetching has the additional benefit of speeding up the computation since all the data is already in the cache. It is impractical if large amounts of data are required for the calculation, e.g. in machine learning applications.

We provide a simple example scenario in Figure 5.2 to illustrate prefetching. The victim runs the function `victim_function`, which executes some operations before performing a secret-dependent access to a table. If `table` spans several cache lines, the attacker can now infer information about the secret by observing which cache lines are accessed by the victim process. In the prefetch protected version, the entire table is loaded into the cache in line 3 before the secret dependent access is performed in line 4. Thus, each cache line spanned by the table is accessed at least once, independently of the secret.

We look at an unprotected AES T-Table implementation as a more realistic example. The AES algorithm consists of four steps: SubBytes, ShiftRows, MixColumns, and AddRoundKey. In the first step, every byte of the plaintext is substituted with another

```
1:
2: function VICTIM_FUNCTION
3:     ⋮
4:
5:
6:     load table[secret]
7: end function
```

```
1: function VICTIM_FUNCTION
2:     ⋮
3:     load table
4:     load table[secret]
5: end function
```

Figure 5.2: Example for (a) vulnerable and (b) protected algorithm. Figure taken from [22]

byte. The way the bytes are substituted is described in the S-Box, which is key-dependent. As the S-Box only depends on the key, but not on the plaintext or round key, it can be pre-computed to save time during the encryption. Another way to speed up AES is using T-Tables. These are precomputed tables that combine the SubBytes and MixColumns operations. T-Tables are lookup tables that combine the S-Box values with every possible column permutation. which can, again, be precomputed. Both S-Box and T-Table implementations use key-dependent pre-computed values and are thus vulnerable to side channel attacks if they are not implemented in constant time.

In an unprotected implementation, the attacker can remove one line of the T-Table from the cache either by flushing or by priming the cache, then wait for the encryption to run. After that she tests for the presence of the line in the cache. The probability of using a specific line of the T-Table is around 92%, increasing with the number of rounds R in the AES algorithm, which depends on the key size. It is given as $(1 - (16/256))^{R*4}$. Thus the information about the actual utilization of the observed line leaks information about the secret key. In turn, the probability of not using one line of the S-Box implementation would be 10^{-20}. However, in the case of the AES implementation of OpenSSL, the probability is reduced to zero since the S-Box is fully prefetched before the execution of each round [19]. As a result, they later conclude that this implementation should not be vulnerable to cache attacks.

Prefetching does not actually remove leaky patterns from the code, it just decreases the window an attacker has to detect a secret dependent access. However, applications protected this way are still considered secure against user-level cache attacks for the following reasons:

- If there is shared memory, a *Flush+Reload* attack monitoring the cache will detect the presence of the data in the cache with high accuracy, but it lacks a

mechanism to infer whether this observation is due to the prefetch or to the actual utilization of the data.

- In absence of shared memory, a *Prime+Probe* attack will require to probe the whole eviction set to evict the prefetched data, which may take longer than the execution of the protected application.
- A user level attacker has no way to control when the victim code executes, thus making it very difficult to synchronize attack and victim code.

In summary, the *temporal resolution* of known attacks is too low to detect an access pattern, despite the pattern still being there. However, theoretically, the protected code still leaks information. We found a way for a regular, unprivileged attacker to exploit prefetch protected code. We provide an analysis on a synthetic benchmark, then show two real-world attacks on an AES and RSA implementation.

5.3 Challenges for the Attack

When attacking unprotected implementations via cache side channel, the exact point of measurement is not crucial. It is sufficient to remove the target data from the cache before the victim starts executing, then probe for the target data during the victim process or after it terminates, depending on the target. Getting close to the start and termination point is favorable as it reduces noise, but not critical. When prefetching is applied, timing is much more important. The attacker needs to evict the data precisely *after the prefetch*, but *before utilization*. In Figure 5.3, we annotated the toy example from Figure 5.2. The crucial part is to evict the data between the prefetch in line 4 and the utilization in line 6.

```
1:                                  1:                  ▷ Eviction target T1
2: function VICTIM_FUNCTION         2: function VICTIM_FUNCTION
3:     ⋮                            3:     ⋮
4:                                  4:     load table
5:                                  5:                  ▷ Eviction target T2
6:     load table[secret]           6:     load table[secret]
7: end function                     7: end function
```

Figure 5.3: Example for (a) vulnerable and (b) protected algorithm. The eviction targets are marked. Figure taken from [22]

As victim and attacker only share the same machine (see Figure 1.1) but the attacker has no special privileges, there is no synchronization between the victim and the attacker. That means that the attacker does not know when the victim is running the targeted application, nor does she know when data is prefetched into the cache. To get the timing just right and evict the data at exactly the right instant, the attacker needs to tackle the following challenges:

1. Detect the victim's execution of the target algorithm.
2. Determine the state of the target after detection.
3. Calculate the remaining time until the data is prefetched.
4. Evict the target data from the LLC at the desired instant.

We designed a minimal worst-case application that implements based on the example in the pseudocode we have been using so far in Figure 5.3 version (b). We can use it to find out whether an attacker can overcome the challenges and how. The application executes a variable number of `xor` operations (line 3) before prefetching a table spanning four cache lines (line 4). The prefetch is followed by a single access, where a secret bit `secret` determines which of the four cache lines is accessed a second time (line 6). The example scenario is the worst case for an attacker, as only a very small window of opportunity arises: the secret-related data is accessed immediately after the prefetch. The attacker has a minimal time window to evict data from the cache to gain information. We created a test setup on an Intel core i5-7600K. The details are summarized in Table 5.1.

Table 5.1: Experimental platform details.

Processor	Intel core i5-7600K
Cores	4
Frequency	3.8 GHz
Inclusive LLC	Yes
LLC slices	8
LLC size	6 MB
LLC ways	12
L1 size	32 KB
L1 ways	8

5.3.1 First Challenge: Detect Execution of the Target Algorithm

The attacker has to detect the execution of the function `victim_function`. Once the execution started, `victim_function` will appear in the cache. We monitor the cache line that contains the call to the function itself. We call this the *target for detection*, or `T1`. We study three different approaches to evict `T1`: *Flush+Reload* *Prime+Probe* and a TSX-based approach.

With the *Flush+Reload* technique, we repeatedly flush `T1` from the cache, wait for around 20 cycles and then reload again. The waiting time needs to be determined empirically on every machine. Once the time it takes to load `T1` indicates it has been retrieved from the cache instead of from the main memory, the victim's execution has been detected and the detection time is saved.

Both *Prime+Probe* and TSX-based detection require an eviction set A mapping to the same sets as the detection target `T1`. We enabled huge pages to construct eviction sets as Liu et al. did in [139]. As discussed in Section 4.3, it is possible to use different approaches that do not require the use of huge pages [105, 221], e.g. via the SPOILER leakage. Assuming that huge pages are enabled in the server simplifies the construction of the eviction sets without changing the attacker model for the actual attack: The way the eviction sets are built does not have an impact on the detection, and it is possible and feasible to build them without huge pages.

Intel's L3 caches implement a pseudo-LRU replacement policy, more specifically a static re-reference interval prediction (SRRIP) policy [25, 220]. Elements in the cache have an age, but the strict order of insertion is not kept and many elements can have the same age. Accessed elements get their age reduced, while non-accessed elements get their age increased. If an element needs to be evicted, the first one with age 3 is evicted. Due to this, always accessing the elements in our eviction set in the same order ensures that we are always about to access the block that the target application will evict as soon as it starts to run. A fixed access order is easily implemented through the pointer-chasing technique, where the data of a cache line contains the pointer to the next cache line: The address of a load cannot be decoded before the previous load is complete. Knowing the eviction candidate also means when using *Prime+Probe*, we can avoid accessing the whole set and just measure the access time after accessing one of the blocks in the eviction set. Accessing only one block is faster and thus increases the success rate. Once we

detect a cache miss we assume the victim has begun its execution, and collect that timestamp.

For TSX-based detection we leverage the fact that, whenever a process is running inside a TSX transaction, the process can either be completed and commit the results or suffer an abort when suffering a L3 cache miss and rollback the computations. The abort was used by Disselkoen et al. to construct a timer-free cache attack (see Subsection 2.9.1 for details). We use the abort as a signaling mechanism instead: We access all the elements in the eviction set during the transaction and, in case it aborts, we consider that the victim process has started and use the abort handler to store the timestamp.

For each method, we collected information from 100,000 executions of the target process. We collect data from both victim and attacker. The victim process starts executing after a random amount of time has elapsed. The attacker process tries to detect the victim process. The results are summarized in Table 5.2. The attacker

Table 5.2: Results for the detection methods: Percentage of correctly detected victim process executions, the mean time until the victim process was detected after it started, and the variance of that time. We require a high detection rate and a low variance.

Approach	Detection rate	Mean	Standard Deviation
Flush+Reload	99.0%	281.1	63.3
Prime+Probe	94.6%	1286.6	367.0
TSX-based	97.3%	320.8	12.4

process correctly detected the execution of the victim process in 99.0% of all cases with *Flush+Reload* in 94.6% of all cases with *Prime+Probe* and in 97.3% of all cases with TSX. As expected, the *Flush+Reload* approach shows great accuracy [79, 193]. However, the results obtained for both TSX and *Prime+Probe* are good enough for detection purposes as well. The executions that were not detected by *Prime+Probe* likely started before the cache was prepared with the eviction set. The undetected executions with TSX have a similar explanation: the victim started executing before while the eviction set was still being loaded inside the transaction.

The detection capability is an important aspect when comparing the detection approaches. But to extract secrets from a prefetch protected implementation, the detection time need to be as constant as possible. A constant detection time makes it easier to determine the state of the victim process after detection, and thus helps

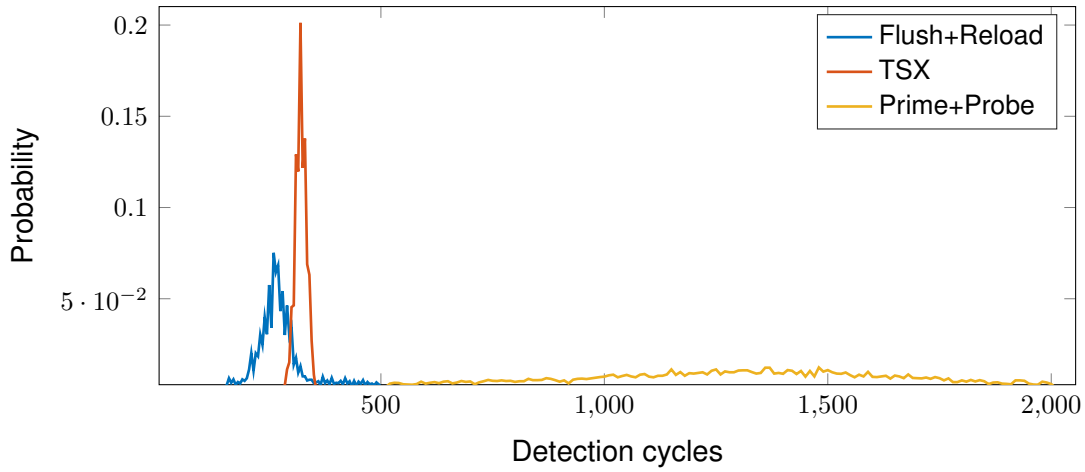

Figure 5.4: Histogram of the times elapsed between the beginning of the execution of the victim process that takes 780 cycles in mean to execute, until it is detected with the different approaches. Figure from [22].

to time the attack to hit the tiny window of opportunity we are aiming for. We thus also analyzed the time elapsed between the start of the victim process and the correct detection. The results are shown in Figure 5.4. The sample application has a very constant execution time of around 600 cycles. When the detection causes a cache miss, the execution time raises to about 780 cycles without increasing its variance. However, the *Prime+Probe* approach usually exceeds the execution time of 780 cycles before the detection is complete, which makes it impossible to evict the prefetched target before it is used.

The *Flush+Reload* approach exhibits a medium standard deviation for detection, which means low reliability for the posterior eviction. Continuously reading from the cache and accessing the eviction candidate may lead to race conditions between the target and the victim process. As a result, the attacker has a hard time to accurately predict how long she has to wait to evict the target: She does not know the exact state of the cache or the exact step of the victim process. We will later see why this is problematic.

In comparison, TSX only shows a slightly smaller detection rate than *Flush+Reload*, but a much smaller standard deviation. Based on these measurements, we state that the TSX technique accurately informs about the execution of the victim and that the attacker receives the abort as soon as the conflict happens. The measured times include the time it takes the CPU to retrieve `T1` from main memory into the

cache. There, it will remove the attacker's data, thus triggering the abort. While a heavy system load leads to additional aborts that are not related to the execution of the victim application, it does not influence the detection time. We conclude that the TSX based method is most suited for our purposes.

Result: The execution of a victim process can be detected most reliably and accurately with a TSX based approach. *Flush+Reload* can also be used.

We thus assume we can use TSX for detection for the remaining experiments and when crafting the attack. As we will see, the TSX-based detection approach by itself solves three of the challenges for the attacker.

5.3.2 Second Challenge: Determine the State of the Target After Detection

Since the victim process continues to execute during the detection, the attacker needs to know the number of operations executed since the beginning of the victim application to determine how much time is left until the prefetch. As the TSX technique generates very prompt aborts, the attacker can profile the victim process execution without an actual victim and measure the clock cycles between loading `T1` and the detection by the attacker. The attacker can subsequently infer the victim's likely activity at the time of detection, as well as the cache contents, including the age of the data.

5.3.3 Third Challenge: Calculate the Remaining Time Until Data is Prefetched

The attacker now knows the target function is executing, and where in the execution the victim is. To tackle the third challenge, she now needs to determine how much time will pass until the prefetch in version (b), line 4. She can again profile the application without an actual victim to find out this *wait time* t.

5.3.4 Fourth Challenge: Evict the Target Data from the LLC at the Desired Instant

Ultimately, the attacker wants to find a way to evict the data after the prefetch. She can then find out whether the victim process reloads the data, and thus retrieve the secret information. As we assume that TSX is used for detection, the attacker's code can continue executing in the abort handler with quite exact knowledge of the cache state. From profiling, the attacker also knows the wait time t. She can now use the abort handler to manipulate the cache and gain information. The challenge is to evict the prefetched data or instructions after the prefetch but before they are used in the execution of the victim process. We call the prefetched data or instructions *target for eviction* or `T2`. We study two different approaches to evict `T2`. Method 1 can be used when there is shared memory between the victim and the attacker, e.g. in form of a shared library. Method 2 works without shared memory. As we will see, the methods differ in the way the cache set is filled during the transaction, in the value of the wait time t (even if the target is the same), in the way this target `T2` is evicted from the cache and finally in the way the information about the actual access to the data is inferred.

5.3.4.1 Method 1 – Remove `T2` by Flushing

Once the victim's access to `T1` is detected, the attacker waits for the wait time t and then evicts the eviction target `T2`. Since there is shared memory, `T2` is simply evicted by using the `clflush` instruction and the information is later retrieved by measuring the time it takes to read it (reload). Remembering Subsection 2.9.1, we see that this is a traditional *Flush+Reload* attack carried out by an abort handler, which allows for very precise timing of the flush instruction: If the prefetch is executed n cycles after detection, then the flush should be triggered at approximately $n - 40$, the minimum value for n being around 60 cycles. The 40 cycles are deducted to ensure the flush can be issued and the address decoded before the prefetched data is in the cache. Note that since the detection target `T1` has to be loaded from main memory, it introduces some variation in the execution time of the victim and, as a result, this last eviction may not be as accurate as the detection. These values were determined empirically and may vary between machines. Just as any *Flush+Reload* attack, method 1 requires shared memory and the availability of the `clflush` instruction to the attacker.

5.3.4.2 Method 2 – Remove T2 with Memory Accesses

There can be various reasons why the attacker cannot remove T2 from the cache by flushing: the clflush instruction may not be available on the platform or in the attackers context, shared memory is disabled or the data has to be retrieved from a non shared variable. It is still possible to achieve the desired eviction accurately by leveraging the replacement policy implemented in the LLC. The replacement policy for 7th generation Intel CPUs described in previous works [5, 26, 220] shows that data is linearly inserted into the cache set and each block is assigned an age value that will change depending on accesses and evictions. For our use case, the following properties are the most relevant:

- Data is evicted from the cache at age 3.
- Only accesses to the LLC update the values of the ages of the elements in the LLC. The age of a block decreases when it is accessed or increases if necessary to select an eviction candidate.
- Once the data is inserted into the cache set, two different ages above age 0 can be distinguished.

To evict T2, the attacker needs to create an eviction set we will call B. In the attack algorithm, the time and order of access for the elements of B depends on two time windows: The time between the access of T1 and T2 and the time between the prefetch of T2 and the actual utilization of it. We will refer to a *large* window if it can fit an entire probe phase of a *Prime+Probe* cycle, and to a *small* window for anything less than that.

We distinguish three different scenarios depending on the window size. They are depicted in Figure 5.5. If the second window is *large*, the size of the first window is irrelevant and a regular $\mathbb{P}$ approach to evict T2 from the cache will suffice. If the second window is small, two different scenarios arise depending on the size of the first window.

The most challenging scenario is the one where the second window is small: It means the attacker has to minimize the time required for the eviction of T2 to achieve an eviction before the utilization. We manage to do so by manipulating the ages of the cache blocks. The method is based on the *Reload+Refresh* attack from [26], but the goal is different. While we use the same technique to age the cache elements, we do not use the known ages of the cache elements to infer a victim

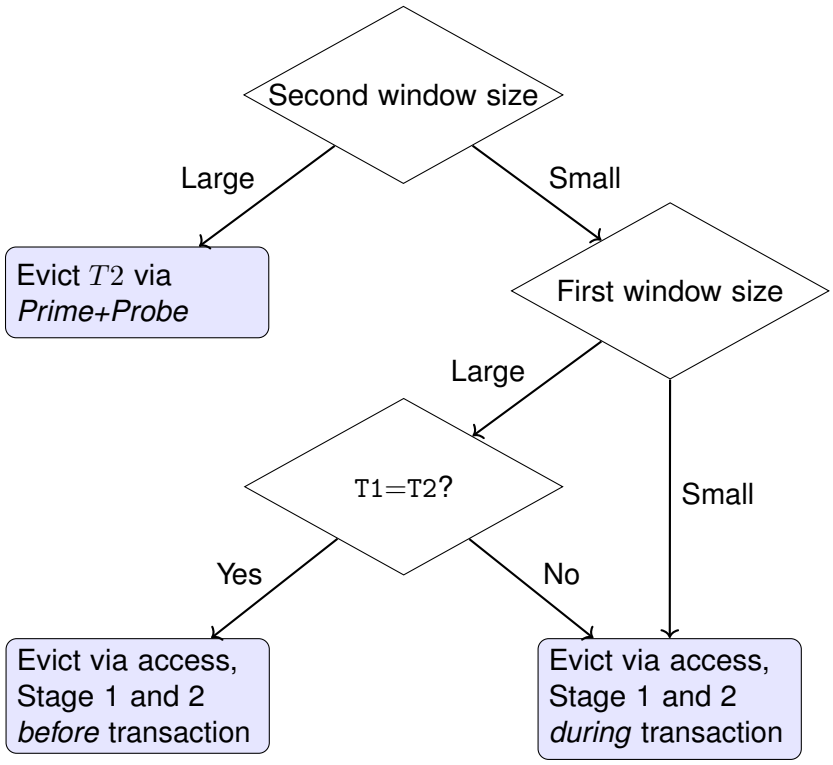

Figure 5.5: Possibilities to evict the target T2 via memory access in case of different window sizes. The first window is the time between the access of T1 and T2, the second the time between the prefetch of T2 and the actual utilization of it. A *large* window can fit an entire *Prime+Probe* circle, a *small* window is anything less than that.

access, but to evict T2 accurately with a single access. The process is depicted in Figure 5.6. The cache is prepared in two stages. The first two take part before the abort:

Stage 1 All elements of the eviction set B (A-H) in Figure 5.6 are accessed to load them into the cache.

Stage 2 The elements B-H are accessed again to change their age, making A the eviction candidate.

Now the attacker waits for the victim process to get to the prefetch of T2. As mentioned before, the time this requires was determined during profiling. The next two steps are used to evict T2.

Stage 3 The victim process prefetches T2. As there is no space in the cache and

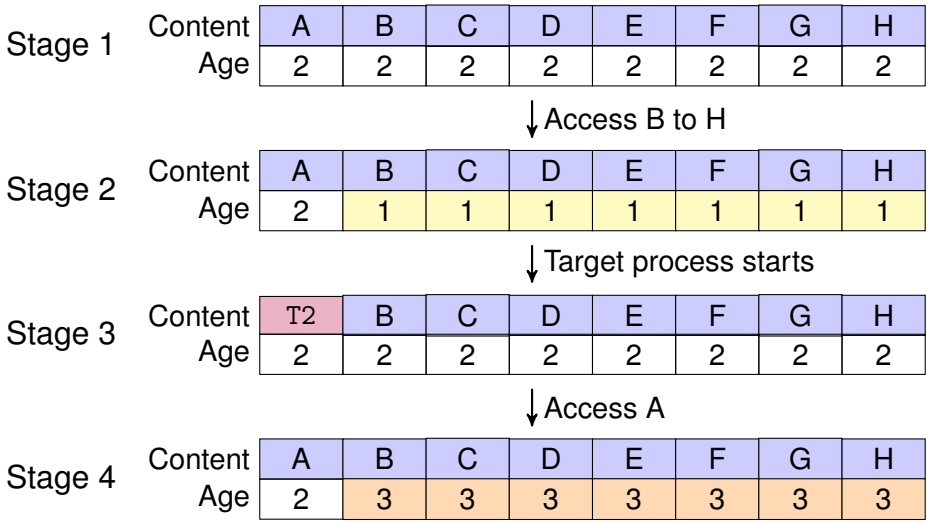

Figure 5.6: Process required to evict the data from the cache with just one access, a technique required for method 2. Figure from [22].

A is the eviction candidate, it is evicted. `T2` instantly becomes the eviction candidate as it is the first entry with the highest age.

Stage 4 The attacker accesses A, thus evicting `T2`.

If the first window is large, stage 1 and stage 2 are performed before the transaction, to avoid detecting `T2` instead of `T1`. A large first window is the preferable scenario for the attacker, as timing is not so critical. If `T1` and `T2` are the same, stage 1 and 2 are performed during the transaction. The same is true if the second window is small. While all the code inside a transaction is executed transiently, the microarchitectural state is not impacted by the rollback in case of an abort. Thus, even though the operations that were performed within the transaction before the abort happened are not visible to the program and the previous logical state is recovered, the changes in the cache remain. After the abort, the attacker waits for wait time t. At that point, the cache is in stage 3. She then evicts `T2`.

We conducted experiments to determine the minimum time elapsed between the execution of `T1` and `T2` in which the attacker is still able to achieve the accurate eviction with just one access. For this, we varied the number of operations executed before the prefetch in our test application to test the different times and evaluate the accuracy of the technique. Note that, even when `T2` has to be loaded from main memory for the prefetch and the execution time increases, the time between

detection and prefetch only changes slightly. The cache state is assumed to be the one depicted in stage 2 of Figure 5.6. We started observing the leakage for times greater than 260 cycles, but the best results were obtained when the second window is larger than 300 cycles. The main reason for the difference between this scenario and the one described in the method 1 is that the eviction is only possible when A has been effectively loaded from main memory (cache miss), whereas in method 1 it is only necessary to wait for the execution time of the flush. Waiting for a load from main memory also means that the attacker should access the block A at time $n - 280$ to achieve the eviction at time n.

During stage 2, we access all elements in the eviction set B except the eviction candidate to change their ages. To avoid pipeline effects, we accessed the elements as a linked list, also known as pointer chasing. As each element needs to be loaded before the next address can be decoded and the LLC usually has higher associativity than the lower level caches, we expect that all data is retrieved from the LLC. As a result, the ages of the accessed elements in the LLC are updated. When measuring the times it takes to read each of the blocks, however, we observed that some of the blocks were retrieved from the L1 cache. We only observed this behavior on in a processor whose LLC is 12-way associative (Intel i5-7600K in Table 5.1), whereas the L1 and the L2 caches are 8-way and 4-way associative respectively. On a different processor (Intel i7-6700K) with a 16-way associative cache, all the data was retrieved from the LLC. We concluded that the L1 replacement policy was responsible for this behavior. It is not documented by Intel, but was described in various publications [5, 220].

As we need to take the L1 replacement policy into account, we will look into it in more details. The policy is illustrated in Figure 5.7.

The tree-PLRU replacement policy starts from the root node and selects a branch depending on the intermediate value of the node. A 0 leads to the selection of the left branch while a 1 makes the algorithm select the right branch. The decision is repeated at every node down the tree until a cache element is selected. If an element in the cache is accessed, all nodes pointing to it are toggled.

Our experiments showed that elements are inserted linearly in the first empty block if the cache is empty, regardless of the value of the nodes. All nodes start with value 0, as depicted in the initial stage of the L1 cache in Figure 5.7 (a). When reading a 12 element LLC eviction set, four elements are evicted from the L1 cache. Intuitively, we would assume the first four elements that entered the L1 cache to be evicted

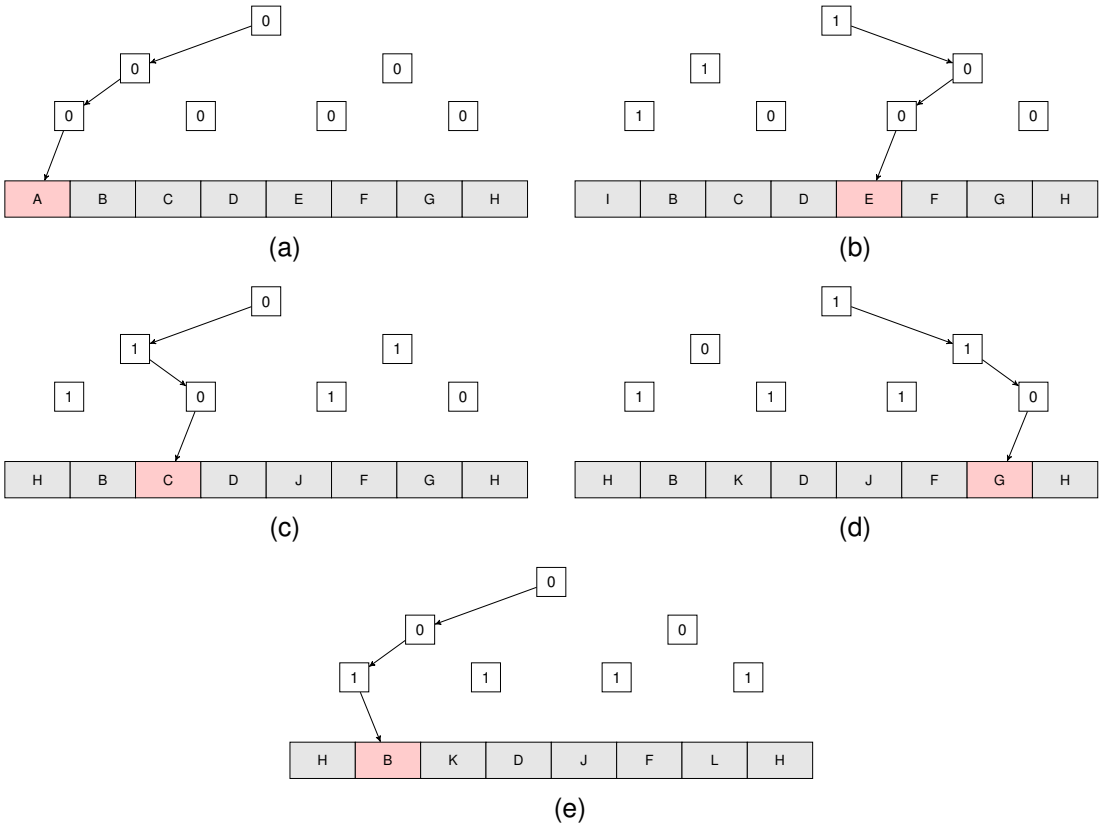

Figure 5.7: Tree structure that controls the Pseudo-LRU replacement policy of L1 and L2 caches. The figure shows an 8-way first level cache. The eviction candidate is marked in red. We assume a 12-way LLC, so 12 elements need to be accessed in stage 1. Figure shows the state after accessing (a) 8 (b) 9 (c) 10 (d) 11 and (e) 12 elements of the eviction set B

first as well. However, as Figure 5.7 shows, the pseudo-LRU replacement policy keeps element B and D in the cache even after reading the whole LLC eviction set (12 elements). If we would access B first in stage 2, it would be retrieved from the L1 cache and it's age in the LLC would not change. With the replacement policy in mind, reordering the linked list with regard to the L1 replacement policy solved the problem.

Result: For evictions with a single access, the lower level cache replacement policies need to be considered when determining the access order of the elements in an LLC eviction set.

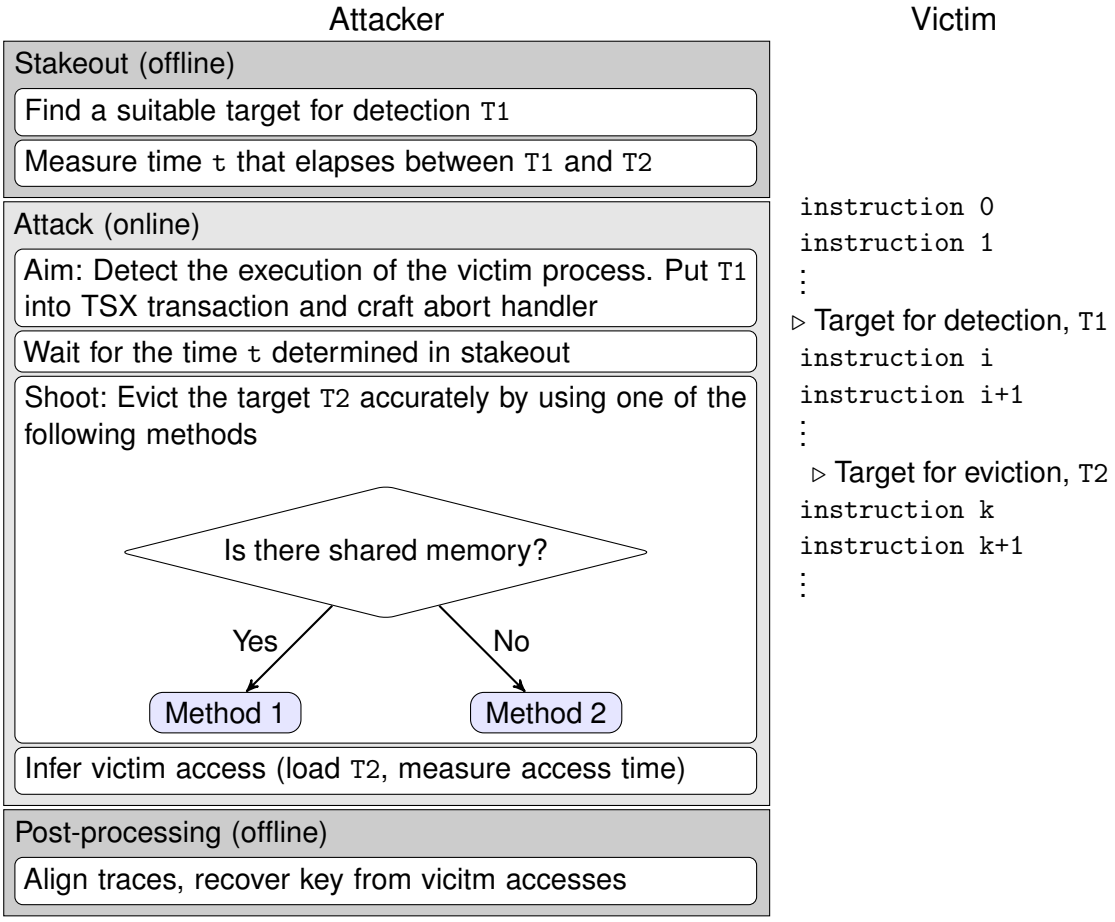

Figure 5.8: Overview of the offline and online attack steps. The online steps are repeated multiple times to collect sufficient traces for key recovery. Figure from [22].

5.4 Crafting an Attack

With the knowledge from the previous analysis, it is now possible to build a full attack. It includes a *stakeout* phase of offline preparation where the attacker has to analyze the victim process. The CACHESNIPER then *aims* at her target in the detection phase and *waits* for a good moment in the execution using the information from stakeout. At the right moment, the attacker *shoots* the correct entry from the cache, evicting it accurately. The attack overview including these steps is depicted in Figure 5.8.

The stakeout phase is similar to the profiling we performed in subsections 5.3.2, 5.3.3 and 5.3.4. The goals are manifold. The attacker needs to

- find an appropriate region of the code that can serve as target for detection `T1`. It has to be accessed long enough before the target `T2` and should not be used too commonly to reduce false positives.
- find set and slice `T1` maps to.
- create eviction set A for `T1`.
- estimate the wait time t for the second attack step.

The wait time t only has to be an estimate as it can be changed dynamically depending on the expected hit/misses ratio and the observations. The stakeout should ideally be performed on a machine similar to the target machine and can be performed offline.

If there is shared memory and `T1` and `T2` can be accessed by the attacker, she can determine the eviction set mapping of the targets herself. Without targets in shared memory, the attacker needs to find the eviction sets by profiling the cache while the victim code is executing. Eviction set finding has been discussed in Subsection 4.3.1 as well as in literature before, and the usual *Prime+Probe* approaches work fine here as well [186]. The information from the aborts can also be used to find the cache set of interest [57].

The waiting time t is the time it takes the victim process to execute the code between `T1` and the point at which the attacker can gain information from evicting `T2` minus the time it takes to remove `T2` from the cache (so either a `clflush` instruction or an access). If the attacker cannot determine t during the stakeout phase, she can still determine it in the online phase of the attack. Determinig t during the online phase is also necessary if the target system behaves differently than expected. She needs to be aware of some characteristics of the victim process that can be observed. For example, around 1% of cache misses will be ideally observed if we hit exactly the last round of the AES encryption or around 50% of the bits are expected to be 1 in an RSA secret key. Thus, the value for t can be retrieved automatically by analyzing the number of hits and misses observed when inferring the victim accesses (line 10 in Figure 5.9) and modifying its value to match the expected values. Of course the attacker can also define an initial value for t and update or adapt it dynamically based on the comparison between the actual and the expected observations. The dynamic approach will increase the number of samples required to derive the secret information.

Input: **Address(T2)**,
Eviction_set A, ▷ Eviction set for T1
t ▷ Waiting time determined in stakeout
Output: $\overline{X}_0$ ▷ Information about the access

```
1: function START_TRANSACTION()
2:     fill_cache_set(A);
3:                                  ▷ Aim: Abort handler detects the access
4:     function ON_ABORT()
5:         time_interrupt=timestamp()+t;
6:         while(timestamp() ≤ time_interrupt) {};
7:         evict_from_cache(T2)                                  ▷ Shoot
8:
9:         Wait until encryption ends
10:        infer_victim_access_to(T2)
11:        if has_accessed(T2) then
12:            X̄0 [t] = 1;                                       ▷ Data used
13:        else
14:            X̄0 [t] = 0;
15:        end if
16:    end function
17:    return X̄0;
18: end function
```

Figure 5.9: Generic attack pseudocode for the TSX-based detection scenario, eviction using method 2, case 2. Figure from [22].

To illustrate, we return to the algorithm shown in Figure 5.3: we would expect to observe 50% of cache hits if the eviction is achieved accurately. If t is too small and the eviction happens before the prefetch, the cache hit rate will be 100%. If t is too large, we only see cache misses as we evict after both prefetch and utilization. To zone in on t, we define an observation window of size w to compute the statistics. Once we have collected the w samples, we observe the hit/misses ratio and either increase or decrease t. The value of t gets stable after some iterations. The number of iterations depends on the size of the window and the original estimation. Unsurprisingly, the best scenario turned out to be the one where the value t is properly selected, and only minor adjustments are made on runtime.

The pseudocode in Figure 5.9 shows the procedure for the online phase of the attack. As shown in Figure 5.8, we generate an eviction set for T1 and determine the waiting time t in the offline phase, then use both as input to our algorithm. The

address of `T2` is an additional input.

5.5 Practical Evaluation

We now have all the prerequisites to recover AES and RSA secret keys from prefetch protected implementations. We not only demonstrate that the implemented countermeasures to prevent cross-core LLC attacks can be circumvented, but also that some previous approaches can benefit from the accurate eviction of the data. All the experiments were performed in the machine described in Table 5.1. Note that the replacement policy on which this attack relies is implemented in the Intel Core processors starting from the 6th generation. The main data for each attacked algorithm are summarized in Table 5.3 and Table 5.4.

5.5.1 CACHESNIPER Against AES

AES [52] is a commonly used symmetric block cipher that operates with data in blocks of 16-bytes. As explained in Section 5.2, it consists of different operations (*AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns*) that are repeated each round. To speed up execution, tables with values that are used repeatedly are precomputed in many implementations, which opens a security vulnerability regarding side channels: an attacker can e.g. profile the cache sets the precomputed tables map to by running an own encryption, then infer victim accesses to extract an unknown key. We analyzed the well-known T-Table implementation and an S-Box implementation. The T-Table is available in OpenSSL 1.0.2k when compiled with the *no-asm* flag, and it is also available in newer OpenSSL versions. If the *no-asm* flag is not used, then the S-Box implementation is used instead. Note that this particular version is the one included by default in our CentOS system, and is shared among all the processes. We have checked that CentOS uses the native OpenSSL implementation for AES.

When OpenSSL is called from the command line, it will use AES-NI for encryption, which is not vulnerable with CACHESNIPER. However, if the C API of OpenSSL is used, a call to `AES_encrypt` would call the S-Box implementation until version 1.0.2k, while a developer wishing to use the default AES-NI instructions (as in the command line) has to use a different instruction to execute the encryption.

Table 5.3: Parameters used for the attack of OpenSSL's prefetch-protected T-Table and S-Box implementations

Parameter	T-Table	S-Box
Detection target `T1`	`AES_encrypt`	S-Box (Prefetch in first round)
Eviction target `T2`	Tei[0]	S-Box (After prefetch in last round)
Samples required method 1	300	≈ 500000
Samples required method 2	360	≈ 500000

5.5.1.1 T-Table-based Implementation

The T-Table implementation uses four tables (T-Table) with pre-computed values of the *SubBytes*, *ShiftRows* and *MixColumns* operations. That is, it transforms these operations into lookup operations in order to improve the performance of the encryption and decryption processes. The accesses to the tables are key dependent and not all of them are used during the encryption process. As data accesses are key dependent, the T-Table implementation is a classical candidate for a cache side channel attack [10, 26, 80, 103]. All of these approaches assume that the attacker monitors the cache before and after the execution of the victim process. That means the attacker can just see that a target cache line, which holds 16 T-Table values, is used during the encryption, but cannot distinguish in which round. This approach generates false positives, and the attacks requires mores samples until eventually one of the key candidates can be clearly distinguished from the others. With our CACHESNIPER technique we can accurately hit the last round and recover the secret key faster. The information we collect is more likely to refer exclusively to the last round. Indeed, while using the same approach to compute the key as previous works [26], we can reduce the samples required to retrieve the whole key from 3000 to 300. The settings for this attack are described in Table 5.3.

Result: We can recover the key of the prefetch protected AES T-Table implementation of OpenSSL with 300 samples. This is an improvement by an order of magnitude.

5.5.1.2 S-Box-based Implementation

As the T-Table implementation is vulnerable to attacks, the S-Box implementation of AES replaced it in many cases. OpenSSL also replaced it, but kept the T-Table implementation anyway for unknown reasons. We suspect backwards compatibility. The S-Box is a table that holds the data for the *SubBytes* operation, concretely 256 byte values. It is used 16 times each round, and the accesses are key-dependent. As a cache line has 64 bytes, one S-Box uses four cache lines. Assuming a key size of 128 bits and as a result 10 rounds, the probability of not accessing one of these cache lines can be calculated by the following equation [19, 28]:

$$\Pr[\text{no access S-Box in encryption}] = \left(1 - \frac{64}{256}\right)^{10*16} \simeq 0 \tag{5.1}$$

Unfortunately for an attacker, it is close to zero, so observing the cache before and after the encryption will not gain enough information to reconstruct the key. However, observing each round individually gives the following probability of not accessing a particular line:

$$\Pr[\text{no access S-Box in round}] = \left(1 - \frac{64}{256}\right)^{16} = 0.01 \tag{5.2}$$

The OpenSSL S-Box implementation includes a prefetch stage before each of the rounds to lower this probability to zero. The encryption process only needs to read one value from each of the four cache lines to ensure that the entire S-Box is in the cache. When the key-dependent memory access is performed, the data will always be in the cache. That means that traditional side channel attacks cannot be used to extract information from this implementation, a conclusion that was affirmed by Irazoqui et al. They used a tool for leakage detection to analyze the OpenSSL code in 2017 and declared it leakage free [102]. The countermeasure only works under the assumption that an attacker cannot interrupt the process at a specific point. CACHESNIPER is capable of exactly this precise interruption.

If the attacker can, however, time the evictions accurately enough, she can evict a cache line between the prefetch and the utilization. With CACHESNIPER, the line

of the S-Box would be `T2`[2]. Just as in the T-Table implementation, an attacker can choose to observe only the last round of encryption, and remove the prefetched data before the last round starts. This even works without shared memory. We can thus perform a cross-core cache attack that recovers the secret key of this protected implementation.

Analog to the T-Table attack, we target the last round. Here, the output of the S-Box is xored with the corresponding round key to get the ciphertext. We assume the ciphertext is known to the attacker. We use a *non-access* approach and thus detect cache misses when the victim did *not* load the data into the cache [26]. This eliminates the problem of finding out which of the 16 S-Box accesses was responsible for a specific part of the S-Box to be in the cache: If we determine that an element has not been accessed, it means none of the operations in the last round has used it. So when we xor each byte of the ciphertext with the 64 values of the S-Box held in the cache line ($k_i = C_i \oplus$ S-Box [0 to 63]), we know that none of these values could be the secret key. This is repeated many times with different ciphertexts, which allows us to infer the key by elimination.

In our test system the last round takes around 40-50 cycles to execute, with a variance of around 15 cycles. To make up for the varying execution times, we adopt the strategy of dynamically updating the value of t described in Section 5.4. Instead of using the probability of 1% as the expected one, we allow up to 7% of cache misses. This way we try to ensure that the observed cache misses actually happen in the round. This value was determined empirically on our machine.

We use the content of one line of the S-Box as `T1` as it is prefetched in every round. Both method 1 (assumes shared memory) and method 2 (does not require shared memory) are able to successfully retrieve the entire secret AES key with a minimum number of samples of about 500 000. Our results show that it is more likely to evict the data in the middle of the execution of the last round than just at the beginning. This means some bytes are recovered faster than others, as it can be observed in Figure 5.10. We can see that the initial four bytes are obtained with 10 000 samples, a relatively small amount. Half of the key bytes has been completely leaked with less than 100 000 samples. Focusing on the different bytes, 12 of the 16 bytes

[2]While analyzing the shared library included in Ubuntu 16.04 or CentOS 7.6 (OpenSSL 1.0.2g), we observed an additional protection. The OpenSSL implementation of AES has four different copies of the S-Boxes. If there are, for example, two processes using the library at the same time, each of them will use a different table. CACHESNIPER can still be conducted.

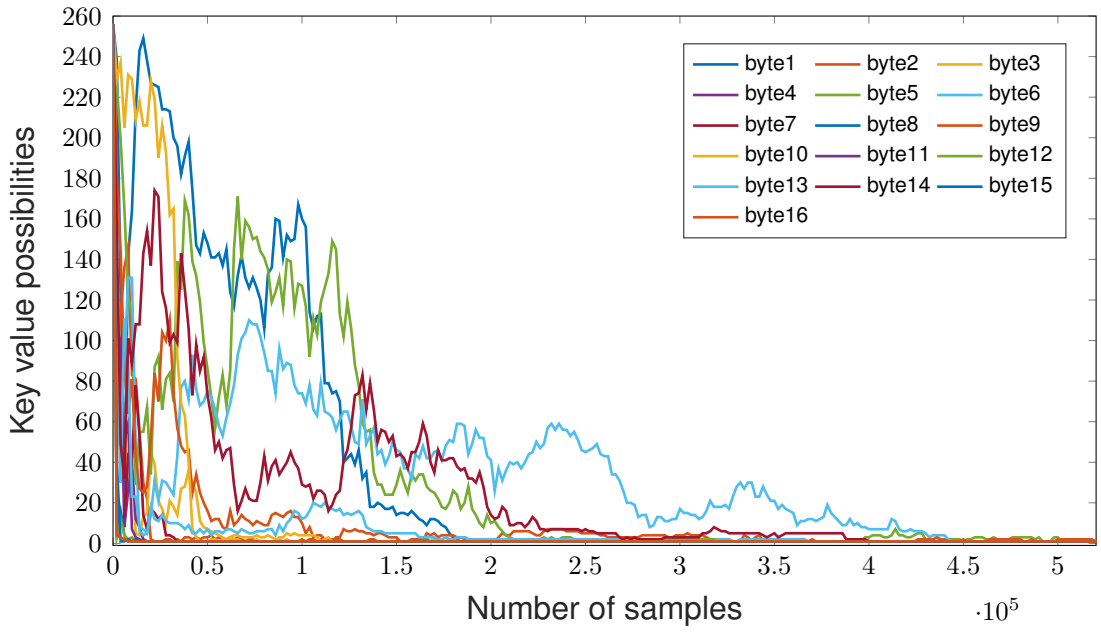

Figure 5.10: Key candidates for each of the bytes of the key of the S-Box AES implementation of OpenSSL retrieved using the TSX-based detection and method 2. Figure from [22]

are already known with 200 000 samples. Retrieving the last 4 bytes of the key is the hardest part, and it requires 300 000 more samples. This shows how difficult it is to evict the data in between the execution of the prefetch and the subsequent access in the last round. While completely brute-forcing these 4 bytes takes 2^{32} trials, the information already collected (with 200000 samples) reduces the key space to around 2^{15} options, making a complete search of the remaining key space faster and stealthier than continuing the attack.

Result: We can recover the key from the prefetch protected AES S-Box implementation of OpenSSL by evicting the target between the last prefetch and the last round of encryption.

Responsible Disclosure We responsibly disclosed our findings to OpenSSL on June 23rd 2020. OpenSSL did not issue a CVE since CACHESNIPER falls outside their threat model [63], which does not include side channel attacks. During our communication with OpenSSL they informed us that they have removed the S-Box from the latest 1.1.1 version. However, we found that in this case the C API

Table 5.4: Parameters used for the attack of the constant execution pattern protected RSA implementation of wolfSSL, attack results and success rates

Parameter	Value
Detection target `T1`	*multiply* or *reduce*
Eviction target `T2`	R[0]
Precision method 1	-
Precision method 2	87,6%

calls instead use the even more vulnerable T-Table implementation. Eventually the communication stopped after we analyzed their proposal of an alternative AES software implementation, which was secure. Despite that, as far as we know, the vulnerable implementations are still part of OpenSSL at the time of writing.

5.5.2 CACHESNIPER Against RSA

RSA is the most widely used public key cryptographic algorithm. It considers a public key (n, e) where n is the product of two prime numbers p and q that remain secret, and a private key (p, q, d) where $d \equiv e^{-1} \pmod{(p-1)(q-1)}$. Only the encryption and decryption operations are relevant to understand the attack. For a message m, the ciphertext c is obtained as $c = m^e \pmod{n}$ and it is recovered with an analogous operation $m = c^d \pmod{n}$. The decryption, which is the exponentiation operation using the secret key d, is the attack target.

There are multiple ways of implementing this exponentiation [74, 120]. We will focus particularly on the square-and-multiply exponentiation, since the wolfSSL implementation is based on it. The square-and-multiply approach scans the bits of the secret exponent d, performing a square operation independently of the value of the scanned bit, and a multiplication if such bit is equal to 1. Thus, an attacker monitoring these operations can retrieve the sequence of bits of the exponent.

The modular exponentiation executed for the RSA decryption operations in the wolfSSL implementation is a variation of this well-known square-and-multiply algorithm. It is shown in Figure 5.11. The countermeasures wolfSSL has deployed to protect this implementation are to always perform the square and the multiply operations for each bit of the exponent (lines 3 and 6) and to load the two possible values of the secret bit related parameter R (R[0] and R[1]) into the cache, so they prevent an

Input: base b, modulo m, exponent $e = (e_{n-1}...e_0)_2$
Output: $b^e \pmod m$

```
1: init(R);
2: for i from n − 1 downto 0 do
3:     mul(R[0], R[1], R[e_i]));                    ▷ Load R[0] and R[1]
4:     modRed(R[e_i]);
5:                                  ▷ R[2] is a temp variable that avoids the leakage of R[e_i]
6:     sqr(R[2], R[2]);
7:     modRed(R[2]);
8: end for
9: return R;
```

Figure 5.11: wolfSSL exponentiation implementation

attacker from distinguishing which one (0 or 1) was actually used during execution of the multiplication function (line 3). For the square operation, they even initialize a temp variable R[2] to hide accesses to (R[0] and R[1]). These are clearly to prevent cache attacks, which can be seen in the source code comments and the release notes [17, 175].

Despite the always-load countermeasure, there are two possible windows to retrieve the secret information. Firstly, at the end of the multiplication operation in line 3 only the result referring to the actually used bit is stored. In that copy process, one of the two possible values is loaded while the other one remains untouched. This leaks the key bit. The second window is even bigger, because of the reduce operation in line 4 that only uses the information of the actual key bit value, not taking the precaution of loading both values. This means that this function could be even vulnerable to a traditional cache attack, although the synchronization between the attacker and the victim process would be a challenge.

Based on the code wolfSSL provides for the tests, we generated different secret keys of 2048 bits, and embedded them in our server application in such a way that it decrypts the received data by calling to the wolfSSL RSA decrypt operation. We can observe the leakage by monitoring accesses to one of the two array entries R[0] or R[1] (our `T2`), since the accesses to each of them to depend on the key bit value (0 or 1). During the execution of the multiply function, they are both loaded into the memory, but at the end of this function they perform a copy operation which only accesses the required value. That is, an attacker can, for example, remove R[0] from the cache before the execution of the copy operation and check it afterwards.

This operation takes around 70-80 cycles in our system, which is enough for the observation.

Attacking this implementation is eased by the reduce operation executed after the multiply operation. This function only loads the correct value of R[e_i], so either R[0] or R[1]. The execution of the reduce operation takes about 2300 cycles in our test system. This time even allows the execution of a complete probe cycle, so we do not have to be so precise evicting the data when targeting it.

We used both the multiply and the reduce operations for the detection `T1` and later evict R[0] (`T2`). Note that, while the functions are shared, R[0] and R[1] are not, so method 2 is required. The attacker has to profile the application to determine t and to find the cache set in which R[0] is loaded. Since our scenario is a continuously running process in a server, the location of R[0] does not change. The task of profiling is eased with the help of the detection of the multiply function. In our posterior experiments we assume that the attacker already knows in which set R[0] is located.

There are other differences to the approach taken in the attack against the S-Box (method 2, case 1). Loading the data of a whole eviction set conflicting with R[0] in the transactional region to achieve a very accurate eviction leads to false positives in detection. This is due to other elements being loaded into the cache set during the large time window of 2300 cycles. This large window also means we do not require such high accuracy, so we can just load some blocks in the transaction to reduce the time it takes later to evict R[0]. This avoids loading the whole eviction set during the transaction. After the detection, only the remaining blocks of the eviction set need to be accessed in the abort handler to retrieve the information about the access (inference step in Figure 5.9, line 10). Thus the attacker still controls the state of the cache for the eviction while avoiding false positives in the detection.

The retrieved key bits depend on both the accuracy of the detection and on the ability of the attacker to remove the data from the cache during the execution of the leaky parts of the code. The mean time between the execution of two multiply operations is about 24000 cycles. That time seems to be “constant” and it is enough for carrying the detection, eviction and retrieving the data. We collected information for the execution of 100 RSA decryptions. Our attack correctly detected 96.8% of the multiply operations introducing 1.3% of false positives. From those correctly detected operations, the information referring to the access to R[0] featured 91% of true positives rate and 87.2% of false negative rate, namely a precision of 87.6%.

Note that no further processing of the results was done. Since we get quite exact timestamps from the TSX aborts, trace alignment becomes fairly easy. For the same reason, some of the retrieved samples that do not match the expected temporal pattern can be discarded to improve the accuracy. Finally, the decision about the correct value of the secret bits of the exponent can be made based on the information retrieved from various traces [151].

Result: Despite an always load strategy, we can recover an RSA key from the wolfSSL implementation with 100 samples. The process is eased by a second implementation fault.

Responsible Disclosure We responsibly disclosed to wolfSSL on June 22nd 2020. WolfSSL immediately issued CVE-2020-15309 and proposed a fix for the vulnerability, which we tested and acknowledged. It was part of the wolfSSL release 4.5.0.

5.6 Countermeasures

The presented attack is feasible due to the fact that code with secret dependent access patterns exist. In order to prevent this leakage, these susceptible patterns must be removed from the source code and the code should be redesigned. In order to help developers to find leakages in their code, there are tools that detect these leakages [227, 228]. As mentioned above, these tools need to be handled with care, as the very OpenSSL implementation attacked in this work was declared leakage free after such an analysis [102]. We showed that merely minimizing the windows of opportunity by prefetching data in the cache is not a hinderance for an attacker. There is still an interval between that prefetch and the actual utilization of the data (as short as the execution of a single instruction) in which an attacker can evict it. Therefore, the attacker has the possibility to observe such accesses and retrieve the secret information.

For AES in particular, efficient and constant-time implementations are possible by using the bit-slicing technique [112]. Alternatively, each S-Box lookup could access all four cache lines and choose the correct lookup value via arithmetic, eliminating cache line leakage. As mentioned above, OpenSSL also provides a constant-time AES software implementation based on bit-slicing, which needs to be selected

via the *-DOPENSSL_AES_CONST_TIME* flag. However it would be good to have widely used APIs call secure implementations. Many deployed applications will update the library, but keep the API calls. Keeping insecure algorithms behind the API violates the security by default policy. Even if developers are aware of microarchitectural side channels and check the code, they may not notice which implementation is used by the API and thus get a false sense of security.

The wolfSSL RSA implementation can be repaired by loading the leaky data into the cache in the two vulnerable functions or use of a temporary value, which is in sync with the currently implemented countermeasures in other parts of the code. Note that the fix will prevent exploitation through the LLC, while it may still be able to retrieve some information in the powerful SGX scenario. The countermeasure was implemented in August 2020.

There are some other approaches intended to defeat cache attacks [67]. Hardware based countermeasures that prevent cache attacks by means of new cache designs [118] or applying hardware modifications [138, 225], can be effective for the presented attack. However, they are not available yet and some of them are not expected to be implemented soon. Similarly, techniques that allocate the victim and the attacker data in different and mutually exclusive cache sets [137] would prevent this attack.

The TSX-based defense CLOAK suggests to perform the entire encryption within a transaction, which would then abort in case of a cache eviction [75], preventing the leakage and CACHESNIPER. When detecting the eviction of the prefetched data this method stops the process and restarts it. However, this method is not widely adopted since it is prone to many false positives, due to spontaneous aborts. Frequent restarts introduce a large overhead and open the door to denial of service attacks. Besides, *Cloak* does not prevent an attack on the L1 cache, if the prefetched data does not belong to a write set.

Finally, detection-based countermeasures monitor the execution of the algorithms they aim to protect. They collect information about execution times or from performance counters (i.e. cache misses or accesses) to detect changes in the execution trace, which could imply an attack [23, 27, 43, 241]. CACHESNIPER was not designed to be stealthy and it generates cache misses on the victim algorithm. However, as we demonstrated by attacking the T-table implementation, CACHESNIPER can also improve the efficiency of existing attacks, seriously limiting the capability of the detection-based countermeasures to trigger the alarm in time.

5.7 Related Work

The AES T-Table implementation is probably one of the most widely attacked implementations [10, 26, 41, 57, 70, 80, 82, 91, 103, 135]. Usually either the first or the last round are targeted, since these rounds only perform an XOR operation between some data and the secret key, which makes key recovery fairly easy. Other approaches, such as targeting a deeper layer implementation of T-Table AES used only to encrypt seeds for the pseudo random number generator in AES, are possible but much less popular [48]. T-Table implementations are widely spread over various libraries and platforms. The fact that prefetches cannot protect the T-Table implementation of OpenSSL from an attacker that works with the eviction policy of the L1 Cache was also described again by Lee et al., who performed the same attack as us and named it *Prime+Retouch* [127].

As it is so vulnerable, the T-Table implementation in OpenSSL was replaced by an S-Box implementation, which after suffering some attacks was protected with a prefetch. Cache-level attackers can only observe accesses under very controlled conditions. C. et al. launch up to 200 spy threads to ensure that the encryption is interrupted after only executing a few instructions [31]. A similar effect can be achieved by using SGX and single-stepping through the victim code, interrupting the enclave after every step [30]. Single-stepping is used by Moghimi et al., who also assume a powerful adversary with full OS control that allows them to monitor the entire L1 data cache [156]. The ability to stop the execution at will and to collect multiple samples per round allows them to distinguish the prefetching stages from the usual operations of the encryption round. CACHESNIPER does not interrupt the victim, requires no elevated attacker privileges and works across cores.

Just as AES, RSA has been a target of side channel attacks for many years [166]. The execution time is much longer and not divided into rounds, which makes prefetching before the execution or at regular intervals during impractical. Instead, always load/always execute strategies are used. When combined with side channel aware code design, such as in the Montgomery reduction with constant execution flow, this strategy is supposed to ensure protection from cache attacks. But matters out of the control of the programmer (e.g. JIT interpreters) can still enable attacks [68].

We use TSX as an enabler for our approach, but are still able to launch a CACHESNIPER attack without it. TSX proved to be a large security risk for Intel: It can be

used to amplify attacks, break kernel space address layout randomization (KASLR), or extract cryptographic secrets [57, 108, 230]. Disselkoen et al. attack an, at that time, unprotected AES T-Table implementation in OpenSSL to show that removing timers is not a sufficient defense against side channel attackers. While they assume an attacker with the same capabilities as we do, they only detect whether the victim used specific data *at any point* during the victim algorithm [57], which is not sufficient to attack a prefetch protected implementation. While CACHESNIPER also relies on the transaction abort as soon as a data loaded within the transaction is evicted from the LLC, we additionally show that the time between the eviction and the abort is almost constant. We can thus determine the state of the victim process with high precision. We also use the abort handler to run the attack, while it just serves as a canary in the work of Disselkoen, Kohlbrenner, Porter, and Tullsen. Other works have used asynchronous aborts to partially leak an RSA key or other data, but assume more powerful attackers [157, 192].

As mentioned above in Section 2.8, Intel issued a microcode update in 2021 that disabled TSX by default in all devices. However, attacks featuring SGX are still present in research up to this day: in 2024, Chowdhuryy et al. used SGX to amplify a replay channel allowing them to exploit a power side channel to infer speculatively accessed secrets [47].

5.8 Summary and Conclusion

We developed CACHESNIPER, a method to exploit very small windows of opportunity with a cache attack. It is noteworthy that while our attacker model includes intimate knowledge of the inner workings of replacement policies and requires profiling of the target algorithm, it is conducted by an unprivileged attacker from user space. The attack works in a cross-core setting. The victim process is not interrupted.

The attacker performs a *stakeout* phase before the attack by profiling the victim process. She then *aims* by detecting the execution of the target algorithm. The information from the stakeout helps her to determine the state of the target algorithm once she detects it. She then *waits* until her window arises, then *shoots* the target address from the cache by a fast eviction. We discuss different approaches for every stage and quantify the resolution an attacker can achieve. We thereby show that several countermeasures against cache attacks do not work as intended.

Prefetch based countermeasures seem like a win-win solution at first glance: Everything is already in the cache when the algorithm starts executing, which may improve runtime, and side-channel resistance is granted on top, practically for free. Unfortunately, the reality looks different. We showed that with exact synchronization with the victim algorithm, an attacker may still recover secrets via the cache even after prefetching was implemented. The attacker can in fact evict the target data between the prefetch and the actual utilization. She can then find out whether there was an actual access. We demonstrated this by attacking a prefetch protected T-Table and S-Box implementation of AES in OpenSSL.

A similar approach is the always load strategy: If there are various alternatives of a value depending on a secret, all of them are loaded. However, an always load strategy only works if great care is taken to use all loaded values in the same way afterwards. We show this by attacking an RSA implementation that uses the always load strategy, but only uses the actual secret in a subsequent reduce operation. By evicting the secret value between the load and the utilization, we can recover the key of the wolfSSL RSA implementation in 100 samples.

We demonstrated that a determined attacker with accurate timing can still attack supposedly protected implementations, even if they were deemed leakage free by tools. Our finding are further proof that only truly constant time, constant execution flow code is a countermeasure for side channel attacks. Prefetching and always load strategies do not produce such code. To ensure developers do not gain a false sense of security, detection tools should be altered to account for the possibility of attacks in very small windows.

6

Conclusion

In the introduction, we presented a scenario in which two individuals, Alice and Bob, unknowingly share a remote machine, highlighting the prevalence of this phenomenon among end users. It is a common misconception that users are aware of the implications of sharing hardware resources, querying cloud-based services, or processing data in the cloud. In reality, many individuals are oblivious to the fact that their activities are executed on shared infrastructure, and that their data is being processed in remote data centers, or even used, for example to train ML models. This lack of awareness raises significant concerns regarding the security and privacy of end users. Educating end users and developers about the dangers of shared hardware and the importance of secure practices, with the goal of promoting a culture of awareness and responsibility, is one way to enhance security. Another possibility is Security by Design: Providing secure solutions by default, where the focus is on designing and implementing secure systems that protect user data and prevent unauthorized access, without requiring users to take additional steps to ensure their security. Unfortunately, the focus is currently more on performance and cost efficiency.

It is my intention to make a tangible impact on the real world through this thesis. The attacks demonstrated here are user-level, and I aim to provide protection for clients by default, without requiring any additional effort from them. To this end, my research is focused on real-world libraries, and I have disclosed all findings in a responsible manner, ensuring that my work can be leveraged to improve the security of widely used systems.

Let us revisit the three primary research goals from Chapter 1. The first objective was the development of privacy-enhancing techniques for outsourced computation. We chose a machine learning scenario in which inference is conducted on an untrusted server containing a TEE and an FPU. With CARNIVAL, we worked on the input and output privacy in this scenario. We build on the SLALOM framework from

Tramer and Boneh and kept the split of the workload between the TEE and an FPU. We re-designed the preprocessing phase to enhance resource utilization while keeping the concept of masking: CARNIVAL generates many masks in public on the untrusted FPU. Making use of the *Subset sum problem over finite fields*, private masks are then generated within the TEE. We prove that CARNIVAL is secure under the longstanding cryptographical assumption that the Subset sum problem over finite fields is a one-way-function. By integrating CARNIVAL into SLALOM we build the variant SLALOM AT THE CARNIVAL (S@C), which includes the improved preprocessing phase. In summary, I present the following results:

- I identified that the preprocessing phase of the SLALOM framework is ineffective.
- I improved it by using the FPU instead of the TEE by using the Subset sum problem over finite fields.
- We obtain a cryptographically secure protocol by using $n \geq 453$ possible summands.
- Experimentally, we show that it is possible to generate a uniform mask distribution with far fewer summands, as few as 65 in the concrete case of the SLALOM.

By building on an existing framework to further improve it, we brought SLALOM one step closer to practical use. The reason the SLALOM framework could be improved so much is that it contains a vastly inefficient preprocessing phase which is not discussed in the corresponding paper. There are many other examples of not including the preprocessing phase in the performance analysis of a framework. We think that it would improve the real world applicability of research results if that changed. An honest analysis of all necessary steps should be the goal to fairly compare new approaches. In the case of SLALOM, this would mean a) measuring the runtime of the offline phase and b) discussing and measuring what happens if not enough masks were produced in the offline phase. In both cases, CARNIVAL increases performance manyfold.

There is no extra effort for the client, and the server owner can work with a speedup of 2.2 to 72.6 in the preprocessing phase. We did not provide an implementation for CARNIVAL or S@C, which is a valuable direction of future work. An implementation would also involve modernizing or re-implementing the SLALOM framework to get a maintainable product. That could lead to a real world use of both frameworks.

As we discuss later, TEEs do not offer side channel resistance, so a potential implementation should keep that in mind and use a constant time algorithm for the summation.

The current technological development in TEEs goes towards larger TEEs, a trend from which CARNIVAL could benefit. If, for example, more masks can be held within the TEE during the addition, delays due to swapping could be avoided.

We also worked on a second new approach on outsourcing computation by leveraging garbled circuits. It is called DASH and is discussed along with the state of the art of outsourced computation and the related work. DASH focuses on usability, providing convenient ways to load models and data into the framework. These qualities make it easy to incorporate DASH into real-world products, providing the clients with more privacy.

The second research objective was the analysis and quantification of microarchitectural behavior that allows an attacker to improve on existing attacks or develop new attacks. When running on the same hardware, processes share resources such as the higher level caches, various buffers, the page table, the GPU or just the physical memory among others. That means they also share a microarchitectural state which can be observed across context switches, thus constituting a microarchitectural side channel. We looked at the shared *store buffer* as well as components for *address resolution* such as the *memory management unit*. While analyzing the store buffer, we discovered *load* instructions are executed *speculatively* without resolving the address fully. When a potential dependency is discovered between the addresses of the load and a *store* in the buffer, a significant delay in executing the *load* instructions can be observed if the store buffer has been filled before. We call this the SPOILER effect. Analyzing the delay leaks partial physical address information, which is usually not available to user space processes. We thus discovered a novel microarchitectural leakage which reveals critical information about physical page mappings to unprivileged processes, thus posing a significant security risk. The leakage can be observed from virtual and sandboxed environments.

In summary, we presented the following results:

- An analysis and explanation of the dependency resolution logic for *store forwarding* in Intel processors, which was previously undocumented. The key observation is that false dependencies in the store buffer generate a high latency in executing load instructions.

- The latency can be observed if 20 bits of the physical address overlap, thus leaking 8 bits of physical address information, which poses a security risk.
- Using pools of aliasing addresses improves the time required to completely cover the cache with eviction sets from 46 seconds to 12 seconds while improving the success rate to 100%.
- The SPOILER effect can be used for other purposes, such as Rowhammer attacks, as a covert channel or to slow down processes.

In recent years, several high-profile "god mode" attacks have been demonstrated, often with the intention of showcasing the vulnerabilities of a particular system or implementation. However, these attacks frequently rely on unrealistic attacker scenarios, where the adversary is assumed to possess elevated privileges, control over the machine, or other extraordinary capabilities. This approach can be misleading, as it fails to account for the vast majority of real-world attacks, which are typically perpetrated by low- to medium-level hackers. In contrast, it is essential to evaluate the security of implementations against a more representative set of attacker scenarios, including those with standard user privileges and limited computing resources. This is particularly relevant for end users, who are more likely to be targeted by opportunistic attackers rather than sophisticated nation-state actors. As such, it is crucial to ensure that hardware and algorithms are designed to withstand attacks from both high- and low-skilled adversaries, rather than solely focusing on the most extreme and unlikely scenarios. By adopting a more nuanced and realistic approach to security evaluation, we can gain a more accurate understanding of the vulnerabilities and strengths of a particular system, and develop more effective countermeasures to protect against a wide range of potential threats. This, in turn, can help to improve the overall security posture of a system, and reduce the risk of successful attacks.

The dependency resolution exploited by SPOILER falls into the category of *speculative behavior*. The processor tries to assume what will happen next in the program to save time. Speculative behavior is used in different parts of the processor, such as address resolution (here) or branch prediction, and can be exploited in many ways to gain privileged information [121, 136]. SPOILER can be executed from user space without special privileges as well as from a sandboxed environment. Users are thus vulnerable just by opening a webpage. This extreme vulnerability of the end user shows, again, how important it is to primarily ensure system security and focus on performance afterwards.

The SPOILER attack had a large impact on the security community and was used in several subsequent works [9, 24, 33, 35, 106, 163, 164, 211]. As the root cause of the SPOILER effect is in hardware, it is difficult to mitigate. As in many other microarchitectural attacks, removing the root cause of the observed latency would decrease performance. That does not seem to be an acceptable price for security for many customers of the vulnerable hardware. It is thus to be expected that the SPOILER leakage will be around for a long time.

The third research objective was the analysis of existing countermeasures to microarchitectural attacks and privacy-enhancing mechanisms and their improvement. Many countermeasures have been proposed against microarchitectural side channel attacks. The best way to remove leakage is by writing true constant-time code, where the data access patterns, timing behavior and execution patterns are not secret dependent. Evaluating how well countermeasures work and whether code is truly constant time is a challenging research task. In this thesis, I analyzed implementations protected by the countermeasure of prefetching or always-load-strategies. The idea is to prohibit an attacker from inferring secrets from the cache state by always fetching all data into the cache, independent of its actual utilization. As countermeasures go, this may actually increase performance as all required data is already cached once the program gets to it.

My analysis shows that there are circumstances under which a user-level attacker can retrieve secrets from a protected implementation. An implementation is still vulnerable if a small window of opportunity remains, which may be as small as a single line of code. The claim of targeting a single line of code from user space sound unrealistic, which is why I demonstrated the application of CACHESNIPER on two real world libraries, OpenSSL and wolfSSL.

In summary, I presented the following results:

- An analysis of the circumstances under which a prefetch protected implementation can still be attacked.

- Identification of four challenges for the attacker: *detect* the execution of the victim algorithm, *determine the time window* between the detection of the target algorithm and the utilization of the target data, *evict* the target data from memory at precisely the right moment, *recover* the information about a potential access.

- Different methods for tackling each of these challenges, highlighting which are applicable under which circumstances.
- The end-to-end attack CACHESNIPER is crafted and tested against several noise benchmarks.
- CACHESNIPER is used to retrieve an AES key from a protected T-table and S-Box implementation as well as an always-load RSA implementation.

Just like SPOILER, CACHESNIPER requires no elevated privileges and can be performed by a user level attacker. It works on two common cryptographic libraries, which shows that it is a very real threat.

From my work, several results can be obtained for both future side channel research and the design of secure algorithms. For side channel research, I present four key challenges and discuss the best way to solve them. The most important contribution here is that TSX is a good option for precisely detecting the victim algorithm. When evicting the target, I show that for a fast and precise eviction, the lower level cache replacement policies need to be considered. With knowledge of the replacement policy, the access order of the elements in an LLC eviction set can be determined.

As for the design of secure algorithms, I successfully showed that prefetching and always load strategies are no valid protection against user-level cache attacks. I offer better strategies along with the corresponding implementations to both attacked libraries in my responsible disclosure process. The vulnerability in the wolfSSL library was closed weeks after my finding. The one in OpenSSL remains open until this day as OpenSSL defines side channel attacks as outside their threat model.

OpenSSL is a community-driven project that relies on the contributions of its users and developers. While it may not be a primary focus of research, incorporating side-channel attacks into its threat model would be a valuable initiative, driven by community pressure. This would not only enhance the security of the OpenSSL library but also align with the collaborative spirit of the project. Given that researchers often provide libraries with assistance to address identified vulnerabilities, the effort required from the OpenSSL community would be relatively minimal. By incorporating side-channel attacks into its threat model, OpenSSL can demonstrate its commitment to security and provide a safer and more reliable foundation for its users. This, in turn, would foster a culture of security awareness and responsibility within the community, ultimately benefiting the broader ecosystem of cryptographic libraries and applications.

My work shows several directions for future research. Firstly, several microarchitectural side channel analysis tools deemed the attacked implementations as secure, which stresses that tools for detecting side channel vulnerabilities need to be tested and updated according to the latest research. Secondly, code needs to be truly constant time. While this is hard and may result in performance penalties, it should not even be a question for cryptographic libraries. A secure implementation should be the base line, and all performance benchmarks should run on that baseline. A secure implementation, of course, includes the utility classes, which, as Sieck et al. show, are often wide open targets [201]. Thirdly, offering developers tools and guidelines for writing better code and analyzing and reviewing it has to be a stronger focus in research as well as technology transfer. Wichelmann et al. for example developed a tool that can be integrated into a GitHub build chain [229].

My research has been guided by a commitment to responsible disclosure, and I have worked closely with the developers of affected libraries to ensure that my findings are addressed in a timely and effective manner. By doing so, I aim to minimize the risk of exploitation and maximize the benefits of my research for the broader community.

This thesis contributed to all three research objectives. All developed attacks work with user level privileges, which shows they are real-world threats. All presented attacks have been thoroughly analyzed and presented with their root cause, allowing other researchers to work with the results. Accordingly, I conducted three responsible disclosure processes with companies affected by the findings of this thesis, further highlighting the practical relevance of my findings. As with many research results, both the attack code and the constructive solution from CARNIVAL should be incorporated into larger frames in the future. For example, the JavaScript code from SPOILER could be used to generate an end-to-end Rowhammer attack in JavaScript without unrealistic assumptions. As CARNIVAL covers a highly relevant scenario, it could be included in real world applications where privacy of the input and output data is important. Mitigating SPOILER and CACHESNIPER and implementing CARNIVAL will have a negative impact on performance, but severely increase the security of the programs as well as protecting the confidentiality of the user data. Hopefully, this tradeoff will be considered worthwhile in the future. Users should have access to safe means of processing their data without being security experts versed in microarchitectural side channel. Not even all developers using standard libraries should have to be security experts. It may be time to put privacy and security before performance, and start to make the default case, the

default hardware, the standard API, secure. If we could do that, Alice and Bob could compute happily ever after, even if they share the same hardware.

References

[1] *1. Introduction cuBLAS 12.8 documentation — docs.nvidia.com*. https://docs.nvidia.com/cuda/cublas/index.html. [Accessed 29-04-2025]. 2025.

[2] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.

[3] A. Abel and J. Reineke. "Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation". In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014. DOI: 10.1109/ISPASS.2014.6844475.

[4] Andreas Abel and Jan Reineke. "Measurement-based modeling of the cache replacement policy". In: *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*. 2013. DOI: 10.1109/RTAS.2013.6531080. URL: https://doi.org/10.1109/RTAS.2013.6531080.

[5] Andreas Abel and Jan Reineke. "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence, RI, USA, 2019. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304062. URL: http://doi.acm.org/10.1145/3297858.3304062.

[6] 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*. US Patent 6,378,062. 2002-04.

[7] 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*. US Patent 5,717,882. 1998-02.

[8] Wilhelmina Afua Addy, Adeola Olusola Ajayi-Nifise, Binaebi Gloria Bello, Sunday Tubokirifuruar Tula, Olubusola Odeyemi, and Titilola Falaiye. “AI in credit scoring: A comprehensive review of models and predictive analytics”. In: *Global Journal of Engineering and Technology Advances* 18.02 (2024), pp. 118–129.

[9] Andrew J. Adiletta, M. Caner Tol, Yarkin Doröz, and Berk Sunar. “Mayhem: Targeted Corruption of Register and Stack Variables”. In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. 2024. DOI: 10.1145/3634737.3637638. URL: https://doi.org/10.1145/3634737.3637638.

[10] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. “Wait a Minute! A fast, Cross-VM Attack on AES”. In: *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*. 2014.

[11] Aman Arora, Moinak Ghosh, Samidh Mehta, Vaughn Betz, and Lizy K. John. “Tensor Slices: FPGA Building Blocks For The Deep Learning Era”. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.4 (2022), 46:1–46:34. DOI: 10.1145/3529650. URL: https://doi.org/10.1145/3529650.

[12] Marshall Ball, Brent Carmer, Tal Malkin, Mike Rosulek, and Nichole Schimanski. “Garbled Neural Networks are Practical”. In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 338. URL: https://eprint.iacr.org/2019/338.

[13] Marshall Ball, Tal Malkin, and Mike Rosulek. “Garbling Gadgets for Boolean and Arithmetic Circuits”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016. DOI: 10.1145/2976749.2978410. URL: https://doi.org/10.1145/2976749.2978410.

[14] Mauro Barni, Pierluigi Failla, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. “Privacy-Preserving ECG Classification With Branching Programs and Neural Networks”. In: *IEEE Trans. Inf. Forensics Secur.* 6.2 (2011), pp. 452–468. DOI: 10.1109/TIFS.2011.2108650. URL: https://doi.org/10.1109/TIFS.2011.2108650.

[15] Mauro Barni, Claudio Orlandi, and Alessandro Piva. “A privacy-preserving protocol for neural-network-based computation”. In: *Proceedings of the 8th workshop on Multimedia & Security, MM&Sec 2006, Geneva, Switzerland,*

September 26-27, 2006. 2006. DOI: 10.1145/1161366.1161393. URL: https://doi.org/10.1145/1161366.1161393.

[16] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. “Formal verification of a constant-time preserving C compiler”. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 7:1–7:30. DOI: 10.1145/3371075. URL: https://doi.org/10.1145/3371075.

[17] Jacob Barthelmeh. *wolfSSL (Formerly CyaSSL) Release 3.10.0*. https://github.com/wolfSSL/wolfssl/releases/tag/v3.10.0-stable. 2016.

[18] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. “Semi-homomorphic Encryption and Multiparty Computation”. In: *EUROCRYPT*. 2011.

[19] Red Hat Security Blog. *It’s all a question of time - AES timing attacks on OpenSSL*. https://access.redhat.com/blogs/766093/posts/1976303. 2014-07.

[20] Xavier Bonnetain, Rémi Bricout, André Schrottenloher, and Yixin Shen. “Improved Classical and Quantum Algorithms for Subset-Sum”. In: *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part II*. 2020. DOI: 10.1007/978-3-030-64834-3_22. URL: https://doi.org/10.1007/978-3-030-64834-3%5C_22.

[21] Ernest F Brickell. “Solving low density knapsacks”. In: *Advances in cryptology*. Springer. 1984.

[22] Samira Briongos, Ida Bruhns, Pedro Malagón, Thomas Eisenbarth, and José Manuel Moya. “Aim, Wait, Shoot: How the CacheSniper Technique Improves Unprivileged Cache Attacks”. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. 2021. DOI: 10.1109/EUROSP51992.2021.00051. URL: https://doi.org/10.1109/EuroSP51992.2021.00051.

[23] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. “CacheShield: Detecting Cache Attacks Through Self-Observation”. In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. Tempe, AZ, USA, 2018. ISBN: 978-1-4503-5632-9. DOI: 10.11

45/3176258.3176320. URL: http://doi.acm.org/10.1145/3176258.317632 0.

[24] Samira Briongos, Ghassan Karame, Claudio Soriente, and Annika Wilde. “No Forking Way: Detecting Cloning Attacks on Intel SGX Applications”. In: *Annual Computer Security Applications Conference, ACSAC 2023, Austin, TX, USA, December 4-8, 2023*. 2023. DOI: 10.1145/3627106.3627187. URL: https://doi.org/10.1145/3627106.3627187.

[25] Samira Briongos, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. “RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020-08. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/briongos.

[26] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M. Moya. “Cache Misses and the Recovery of the Full AES 256 Key”. In: *Applied Sciences* 9.5 (2019). ISSN: 2076-3417. DOI: 10.3390/app9050944. URL: https://www.mdpi.com/2076-3417/9/5/944.

[27] Samira Briongos, Pedro Malagón, José L. Risco-Martín, and José M. Moya. “Modeling Side-channel Cache Attacks on AES”. In: *Proceedings of the Summer Computer Simulation Conference*. Montreal, Quebec, Canada, 2016. ISBN: 978-1-5108-2424-9. URL: http://dl.acm.org/citation.cfm?id=3015574.3015611.

[28] Samira Briongos Herrero. “Analysis and design of microarchitectural side-channel attacks and countermeasures”. PhD thesis. Telecomunicacion, 2019.

[29] Ida Bruhns, Sebastian Berndt, Jonas Sander, and Thomas Eisenbarth. “Slalom at the Carnival: Privacy-preserving Inference with Masks from Public Knowledge”. In: *IACR Commun. Cryptol.* 1.3 (2024), p. 40. DOI: 10.62056/AKP-49QGXQ. URL: https://doi.org/10.62056/akp-49qgxq.

[30] 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, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. 2017. DOI: 10.1145/3152701.3152706. URL: https://doi.org/10.1145/3152701.3152706.

[31] Ashokkumar C., Bholanath Roy, Bhargav Sri Venkatesh Mandarapu, and Bernard Menezes. ““S-Box” Implementation of AES Is Not Side Channel Resistant”. In: *Journal of Hardware and Systems Security* 4 (2019-12). DOI: 10.1007/s41635-019-00082-w.

[32] 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: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[33] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. “Fallout: Leaking Data on Meltdown-resistant CPUs”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2019.

[34] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. “Cryptanalytic Extraction of Neural Network Models”. In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*. 2020. DOI: 10.1007/978-3-030-56877-1_7. URL: https://doi.org/10.1007/978-3-030-56877-1%5C_7.

[35] Anirban Chakraborty, Nikhilesh Singh, Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. “Timed speculative attacks exploiting store-to-load forwarding bypassing cache-based countermeasures”. In: *DAC ’22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*. 2022. DOI: 10.1145/3489517.3530493. URL: https://doi.org/10.1145/3489517.3530493.

[36] Nishanth Chandran, Divya Gupta, Sai Lakshmi Bhavana Obbattu, and Akash Shah. “SIMC: ML Inference Secure Against Malicious Clients at Semi-Honest Cost”. In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/chandran.

[37] Yan-Cheng Chang and Chi-Jen Lu. “Oblivious Polynomial Evaluation and Oblivious Neural Learning”. In: *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology*

and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings. 2001. DOI: 10.1007/3-540-45682-1_22. URL: https://doi.org/10.1007/3-540-45682-1%5C_22.

[38] Arunava Chaudhuri, Shubhi Shukla, Sarani Bhattacharya, and Debdeep Mukhopadhyay. "Secured and Privacy-Preserving GPU-Based Machine Learning Inference in Trusted Execution Environment: A Comprehensive Survey". In: *17th International Conference on COMmunication Systems and NETworks, COMSNETS 2025, Bengaluru, India, January 6-10, 2025*. 2025. DOI: 10.1109/COMSNETS63942.2025.10885734. URL: https://doi.org/10.1109/COMSNETS63942.2025.10885734.

[39] Jiahao Chen, Dingji Li, Zeyu Mi, Yuxuan Liu, Binyu Zang, Haibing Guan, and Haibo Chen. "Security and Performance in the Delegated User-level Virtualization". In: *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. 2023. URL: https://www.usenix.org/conference/osdi23/presentation/chen.

[40] Ruiqi Chen, Tianyu Wu, Yuchen Zheng, and Ming Ling. "MLoF: Machine Learning Accelerators for the Low-Cost FPGA Platforms". In: *Applied Sciences* 12.1 (2022). ISSN: 2076-3417. DOI: 10.3390/app12010089. URL: https://www.mdpi.com/2076-3417/12/1/89.

[41] Shing Hing William Cheng, Chitchanok Chuengsatiansup, Daniel Genkin, Dallas McNeil, Toby Murray, Yuval Yarom, and Zhiyuan Zhang. "Evict + Spec + Time: Exploiting Out-of-Order Execution to Improve Cache-Timing Attacks". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024.3 (2024), pp. 224–248. DOI: 10.46586/TCHES.V2024.I3.224-248. URL: https://doi.org/10.46586/tches.v2024.i3.224-248.

[42] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*. 2017. DOI: 10.1007/978-3-319-70694-8_15. URL: https://doi.org/10.1007/978-3-319-70694-8%5C_15.

[43] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. "Real time detection of cache-based side-channel attacks using hardware performance counters". In: *Applied Soft Computing* 49 (2016), pp. 1162–1174.

[44] Joseph I. Choi and Kevin R. B. Butler. “Secure Multiparty Computation and Trusted Hardware: Examining Adoption Challenges and Opportunities”. In: *Secur. Commun. Networks* 2019 (2019), 1368905:1–1368905:28.

[45] Benny Chor and Ronald L Rivest. “A knapsack-type public key cryptosystem based on arithmetic in finite fields”. In: *IEEE Transactions on Information Theory* 34.5 (1988), pp. 901–909.

[46] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. “Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference”. In: *CoRR* abs/1811.09953 (2018). arXiv: 1811.09953. URL: http://arxiv.org/abs/1811.09953.

[47] Md Hafizul Islam Chowdhuryy, Zhenkai Zhang, and Fan Yao. “PowSpectre: Powering Up Speculation Attacks with TSX-based Replay”. In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. 2024. DOI: 10.1145/3634737.3661139. URL: https://doi.org/10.1145/3634737.3661139.

[48] Shaanan Cohney, Andrew Kwong, Shachar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. *Pseudorandom Black Swans: Cache Attacks on CTR_DRBG*. Cryptology ePrint Archive, Report 2019/996. https://eprint.iacr.org/2019/996. 2019.

[49] Victor Costan and Srinivas Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 86. URL: http://eprint.iacr.org/2016/086.

[50] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[51] *CUDA Deep Neural Network — developer.nvidia.com*. https://developer.nvidia.com/cudnn. [Accessed 29-04-2025]. 2024.

[52] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002. ISBN: 3-540-42580-2. DOI: 10.1007/978-3-662-04722-4. URL: http://dx.doi.org/10.1007/978-3-662-04722-4.

[53] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: *CRYPTO*. 2012.

[54] Aritra Dhar, Clément Thorens, Lara Magdalena Lazier, and Lukas Cavigelli. “Guardain: Protecting Emerging Generative AI Workloads on Heterogeneous NPU”. In: *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*. 2025. DOI: 10.1109/SP61157.2025.00221. URL: https://doi.org/10.1109/SP61157.2025.00221.

[55] Abdulrahman Diaa, Lucas Fenaux, Thomas Humphries, Marian Dietz, Faezeh Ebrahimianghazani, Bailey Kacsmar, Xinda Li, Nils Lukas, Rasoul Akhavan Mahdavi, Simon Oya, Ehsan Amjadian, and Florian Kerschbaum. “Fast and Private Inference of Deep Neural Networks by Co-designing Activation Functions”. In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024 (in Print)*. 2024. URL: https://www.usenix.org/system/files/sec24summer-prepub-373-diaa.pdf.

[56] Dave Dice, Tim Harris, Alex Kogan, and Yossi Lev. “The Influence of Malloc Placement on TSX Hardware Transactional Memory”. In: *CoRR* abs/1504.04640 (2015). arXiv: 1504.04640. URL: http://arxiv.org/abs/1504.04640.

[57] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC, 2017. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen.

[58] Javier M. Duarte et al. “FPGA-Accelerated Machine Learning Inference as a Service for Particle Physics Computing”. In: *Comput. Softw. Big Sci.* 3.1 (2019). DOI: 10.1007/S41781-019-0027-2. URL: https://doi.org/10.1007/s41781-019-0027-2.

[59] Soumia Zohra El Mestari, Gabriele Lenzini, and Huseyin Demirci. “Preserving data privacy in machine learning systems”. In: *Computers & Security* 137 (2024), p. 103605. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2023.103605. URL: https://www.sciencedirect.com/science/article/pii/S0167404823005151.

[60] David Evans, Vladimir Kolesnikov, and Mike Rosulek. “A Pragmatic Introduction to Secure Multi-Party Computation”. In: *Found. Trends Priv. Secur.* 2.2-3 (2018), pp. 70–246. DOI: 10.1561/3300000019. URL: https://doi.org/10.1561/3300000019.

[61] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “Jump over ASLR: Attacking Branch Predictors to Bypass ASLR”. In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. Taipei, Taiwan, 2016.

[62] Agner Fog. “The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers”. In: *Copenhagen University College of Engineering* (2012), pp. 02–29.

[63] OpenSSL Software foundation. *Security Policy*. https://www.openssl.org/policies/secpolicy.html. [Online; accessed 26-June-2020]. 2012.

[64] Rusins Freivalds. “Probabilistic Machines Can Use Less Running Time.” In: *IFIP congress*. 1977.

[65] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU”. In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 2018. DOI: 10.1109/SP.2018.00022. URL: https://doi.org/10.1109/SP.2018.00022.

[66] Karthik Garimella, Nandan Kumar Jha, and Brandon Reagen. “Sisyphus: A Cautionary Tale of Using Low-Degree Polynomial Activations in Privacy-Preserving Deep Learning”. In: *CoRR* abs/2107.12342 (2021). arXiv: 2107.12342. URL: https://arxiv.org/abs/2107.12342.

[67] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *Journal of Cryptographic Engineering* 8.1 (2018-04), pp. 1–27. ISSN: 2190-8516. DOI: 10.1007/s13389-016-0141-6. URL: https://doi.org/10.1007/s13389-016-0141-6.

[68] 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*. Springer. 2018.

[69] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy”. In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2016. URL: http://proceedings.mlr.press/v48/gilad-bachrach16.html.

[70] Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss. “Generic and Automated Drive-by GPU Cache Attacks from the Browser”. In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. 2024. DOI: 10.1145/3634737.3656283. URL: https://doi.org/10.1145/3634737.3656283.

[71] Oded Goldreich. *Foundations of Cryptography: Basic applications*. Vol. 2. Cambridge University Press, 2010.

[72] Oded Goldreich and Leonid A. Levin. “A Hard-Core Predicate for all One-Way Functions”. In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA*. 1989. DOI: 10.1145/73007.73010. URL: https://doi.org/10.1145/73007.73010.

[73] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MITP, 2018.

[74] Daniel M. Gordon. “A Survey of Fast Exponentiation Methods”. In: *J. Algorithms* 27.1 (1998-04), pp. 129–146. ISSN: 0196-6774.

[75] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. “Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC, 2017-08. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss.

[76] 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: *CoRR* abs/1710.00551 (2017). arXiv: 1710.00551. URL: http://arxiv.org/abs/1710.00551.

[77] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria, 2016. ISBN: 978-1-4503-4139-4.

[78] 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*. Cham, 2016. ISBN: 978-3-319-40667-1.

[79] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 2016.

[80] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C., 2015. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[81] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice”. In: *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. Washington, DC, USA, 2011. ISBN: 978-0-7695-4402-1. DOI: 10.1109/SP.2011.22. URL: http://dx.doi.org/10.1109/SP.2011.22.

[82] Berk Gülmezoglu, Mehmet Sinan Inci, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. “A Faster and More Realistic Flush+Reload Attack on AES”. In: *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*. 2015. DOI: 10.1007/978-3-319-21476-4_8. URL: http://dx.doi.org/10.1007/978-3-319-21476-4%5C_8.

[83] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. “Lest we remember: cold-boot attacks on encryption keys”. In: *Commun. ACM* 52.5 (2009), pp. 91–98. DOI: 10.1145/1506409.1506429. URL: https://doi.org/10.1145/1506409.1506429.

[84] Lars T Hansen. *Shared memory: Side-channel information leaks*. 2016.

[85] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. “DarKnight: An Accelerated Framework for Privacy and Integrity Preserving Deep Learning Using Trusted Hardware”. In: *CoRR* abs/2207.00083 (2022). DOI: 10.48550/ARXIV.2207.00083. arXiv: 2207.00083. URL: https://doi.org/10.48550/arXiv.2207.00083.

[86] Muneeb ul Hassan. *VGG16 – Convolutional Network for Classification and Detection*. https://neurohive.io/en/popular-networks/vgg16/. Accessed: 2025-01-20. 2018.

[87] Zecheng He, Guangyuan Hu, and Ruby B. Lee. “New Models for Understanding and Reasoning about Speculative Execution Attacks”. In: *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. 2021. DOI: 10.1109/HPCA51647.2021.00014. URL: https://doi.org/10.1109/HPCA51647.2021.00014.

[88] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, p. 78. ISBN: 0128119055.

[89] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N. Wright. “Privacy-preserving Machine Learning as a Service”. In: *Proc. Priv. Enhancing Technol.* 2018.3 (2018), pp. 123–142. DOI: 10.1515/POPETS-2018-0024. URL: https://doi.org/10.1515/popets-2018-0024.

[90] Sebastien Hily, Zhongying Zhang, and Per Hammarlund. *Resolving false dependencies of speculative load instructions*. US Patent 7,603,527. 2009-10.

[91] Gal Horowitz, Eyal Ronen, and Yuval Yarom. “Spec-o-Scope: Cache Probing at Cache Speed”. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. 2024. DOI: 10.1145/3658644.3690313. URL: https://doi.org/10.1145/3658644.3690313.

[92] Wei-Ming Hu. “Lattice scheduling and covert channels”. In: *IEEE Symposium on Research in Security and Privacy*. 1992. DOI: 10.1109/RISP.1992.213271. URL: http://dx.doi.org/10.1109/RISP.1992.213271.

[93] Ralf Hund, Carsten Willems, and Thorsten Holz. “Practical timing side channel attacks against kernel space ASLR”. In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013.

[94] Russell Impagliazzo and Moni Naor. “Efficient cryptographic schemes provably as secure as subset sum”. In: *Journal of cryptology* 9.4 (1996), pp. 199–216.

[95] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. *Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud*. Tech. rep. IACR Cryptology ePrint Archive, 2015.

[96] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cache Attacks Enable Bulk Key Recovery on the Cloud”. In: *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. 2016.

[97] Intel. *Intel Performance Monitoring Events*. 2025. URL: https://web.archive.org/web/20250831211026/https://perfmon-events.intel.com/ (visited on 2025-08-21).

[98] Intel. *Intel Software Guard Extensions Developer Reference for Linux* OS*. 2021. URL: https://download.01.org/intel-sgx/sgx-linux/2.14/docs/Intel_SGX_Developer_Reference_Linux_2.14_Open_Source.pdf.

[99] Intel. *Intel's Lunar Lake processors arriving Q3 2024*. 2024-10. URL: https://newsroom.intel.com/artificial-intelligence/intels-lunar-lake-processors-arriving-q3-2024.

[100] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[101] *Intel® transactional synchronization extensions (Intel® TSX) memory and performance monitoring update for Intel® processors*. https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html. [Online; last accessed 12 Dec 2024]. 2021.

[102] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun K. Kanuparthi, Thomas Eisenbarth, and Berk Sunar. “Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries”. In: *CoRR* abs/1709.01552 (2017). arXiv: 1709.01552. URL: http://arxiv.org/abs/1709.01552.

[103] 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: *36th IEEE Symposium on Security and Privacy (S&P 2015)*. 2015.

[104] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Systematic Reverse Engineering of Cache Slice Selection in Intel Processors”. In: *2015 Euromicro Conference on Digital System Design (DSD)*. 2015-08. DOI: 10.1109/DSD.2015.56. URL: doi.ieeecomputersociety.org/10.1109/DSD.2015.56.

[105] 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: *28th USENIX Security Symposium (USENIX Security 19)*. 2019.

[106] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. “Signature Correction Attack on Dilithium Signature Scheme”. In: *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*. 2022. DOI: 10.1109/EUROSP53844.2022.00046. URL: https://doi.org/10.1109/EuroSP53844.2022.00046.

[107] Eric Jahns, Milan Stojkov, and Michel A. Kinsy. “Privacy-Preserving Deep Learning: A Survey on Theoretical Foundations, Software Frameworks, and Hardware Accelerators”. In: *IEEE Access* 13 (2025), pp. 67821–67855. DOI: 10.1109/ACCESS.2025.3561721. URL: https://doi.org/10.1109/ACCESS.2025.3561721.

[108] Yeongjin Jang, Sangho Lee, and Taesoo Kim. “Breaking kernel address space layout randomization with intel tsx”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016.

[109] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. “ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms”. In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/jattke.

[110] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. “GAZELLE: A Low Latency Framework for Secure Neural Network Inference”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar.

[111] David Kaplan. “AMD x86 Memory Encryption Technologies”. In: Austin, TX, 2016-08.

[112] Emilia Käsper and Peter Schwabe. “Faster and timing-attack resistant AES-GCM”. In: *International Workshop on Cryptographic Hardware and Embedded Systems — CHES*. Springer. 2009.

[113] Aniket Kate and Ian Goldberg. “Generalizing cryptosystems based on the subset sum problem”. In: *International Journal of Information Security* 10.3 (2011), pp. 189–199.

[114] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. “The Gates of Time: Improving Cache Attacks with Transient Execution”. In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/katzman.

[115] Kasem Khalil, Tamador Mohaidat, Mahmoud Darwich, Ashok Kumar, and Magdy A. Bayoumi. “Efficient Hardware Implementation of Artificial Neural Networks on FPGA”. In: *6th IEEE International Conference on AI Circuits and Systems, AICAS 2024, Abu Dhabi, United Arab Emirates, April 22-25, 2024*. 2024. DOI: 10.1109/AICAS59952.2024.10595867. URL: https://doi.org/10.1109/AICAS59952.2024.10595867.

[116] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. “iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. 2023. DOI: 10.1145/3576915.3616611. URL: https://doi.org/10.1145/3576915.3616611.

[117] Taechan Kim and Mehdi Tibouchi. “Bit-Flip Faults on Elliptic Curve Base Fields, Revisited”. In: *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*. 2014. DOI: 10.1007/978-3-319-07536-5_11. URL: https://doi.org/10.1007/978-3-319-07536-5%5C_11.

[118] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. “STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud”. In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. 2012. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kim.

[119] Yoongu Kim, Ross Daly, Jeremie S. 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/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. 2014. DOI: 10.1109/ISCA.2014.6853210. URL: https://doi.org/10.1109/ISCA.2014.6853210.

[120] C.K. KoÃ§. “Analysis of sliding window techniques for exponentiation”. In: *Computers & Mathematics with Applications* 30.10 (1995), pp. 17–24. ISSN: 0898-1221. DOI: https://doi.org/10.1016/0898-1221(95)00153-P.

[121] Paul Kocher, 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: *meltdownattack.com* (2018). URL: https://spectreattack.com/spectre.pdf.

[122] Yuriy Kochura, Yuri G. Gordienko, Vlad Taran, Nikita Gordienko, Alexandr Rokovyi, Oleg Alienin, and Sergii G. Stirenko. “Batch Size Influence on Performance of Graphic and Tensor Processing Units during Training and Inference Phases”. In: *CoRR* abs/1812.11731 (2018). arXiv: 1812.11731. URL: http://arxiv.org/abs/1812.11731.

[123] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. “Half-Double: Hammering From the Next Row Over”. In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double.

[124] 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*. US Patent 9,507,725. 2016-11.

[125] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. “CrypTFlow: Secure TensorFlow Inference”. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020. DOI: 10.1109/SP40000.2020.00092. URL: https://doi.org/10.1109/SP40000.2020.00092.

[126] Jeffrey C Lagarias and Andrew M Odlyzko. “Solving low-density subset sum problems”. In: *Journal of the ACM (JACM)* 32.1 (1985), pp. 229–246.

[127] Jae-Hyuk Lee, Fan Sang, and Taesoo Kim. “Prime+Retouch: When Cache is Locked and Leaked”. In: *CoRR* abs/2402.15425 (2024). DOI: 10.48550/ARXIV.2402.15425. arXiv: 2402.15425. URL: https://doi.org/10.48550/arXiv.2402.15425.

[128] Daniel Lehmann, Johannes Kinder, and Michael Pradel. “Everything Old is New Again: Binary Security of WebAssembly”. In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[129] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. “Muse: Secure Inference Resilient to Malicious Clients”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/lehmkuhl.

[130] Abraham Lempel. “Cryptology in transition”. In: *ACM Computing Surveys (CSUR)* 11.4 (1979), pp. 285–303.

[131] Congwu Li, Le Guan, Jingqiang Lin, Bo Luo, Quanwei Cai, Jiwu Jing, and Jing Wang. “Mimosa: Protecting Private Keys Against Memory Disclosure Attacks Using Hardware Transactional Memory”. In: *IEEE Trans. Dependable Secur. Comput.* 18.3 (2021), pp. 1196–1213. DOI: 10.1109/TDSC.2019.2897666. URL: https://doi.org/10.1109/TDSC.2019.2897666.

[132] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. “Design and Verification of the Arm Confidential Compute Architecture”. In: *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. 2022. URL: https://www.usenix.org/conference/osdi22/presentation/li.

[133] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. “FP-BNN: Binarized neural network on FPGA”. In: *Neurocomputing* 275 (2018), pp. 1072–1086. DOI: 10.1016/J.NEUCOM.2017.09.046. URL: https://doi.org/10.1016/j.neucom.2017.09.046.

[134] Walter Link and Herbert May. “Eigenschaften von MOS-Ein-Transistorspeicherzellen bei tiefen Temperaturen”. In: (1979).

[135] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp.

[136] 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: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD, 2018. ISBN: 978-1-931971-46-1.

[137] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. “CATalyst: Defeating last-level cache side channel attacks in cloud computing”. In: *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. 2016. DOI: 10.1109/HPCA.2016.7446082. URL: https://doi.org/10.1109/HPCA.2016.7446082.

[138] Fangfei Liu and Ruby B. Lee. “Random Fill Cache Architecture”. In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Cambridge, United Kingdom, 2014. ISBN: 978-1-4799-6998-2. DOI: 10.1109/MICRO.2014.28. URL: http://dx.doi.org/10.1109/MICRO.2014.28.

[139] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015. DOI: 10.1109/SP.2015.43. URL: https://doi.org/10.1109/SP.2015.43.

[140] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. “Oblivious Neural Network Predictions via MiniONN Transformations”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2017. DOI: 10.1145/3133956.3134056. URL: https://doi.org/10.1145/3133956.3134056.

[141] Ximeng Liu, Lehui Xie, Yaopeng Wang, Jian Zou, Jinbo Xiong, Zuobin Ying, and Athanasios V. Vasilakos. “Privacy and Security Issues in Deep Learning: A Survey”. In: *IEEE Access* 9 (2021), pp. 4566–4593. DOI: 10.1109/ACCESS.2020.3045078. URL: https://doi.org/10.1109/ACCESS.2020.3045078.

[142] Qian Lou, Bo Feng, Geoffrey Charles Fox, and Lei Jiang. “Glyph: Fast and Accurately Training Deep Neural Networks on Encrypted Data”. In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/685ac8cadc1be5ac98da9556bc1c8d9e-Abstract.html.

[143] Giorgi Maisuradze and Christian Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018. DOI: 10.1145/3243734.3243761. URL: https://doi.org/10.1145/3243734.3243761.

[144] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters”. In: *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*. Kyoto, Japan, 2015. ISBN: 978-3-319-26361-8.

[145] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. “Innovative instructions and software model for isolated execution”. In: *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*. 2013. DOI: 10.1145/2487726.2488368. URL: https://doi.org/10.1145/2487726.2488368.

[146] *Intel 64 Architecture Memory Ordering White Paper*. http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf. Accessed: 2018-11-26. 2008.

[147] Ralph Merkle and Martin Hellman. “Hiding information and signatures in trapdoor knapsacks”. In: *IEEE transactions on Information Theory* 24.5 (1978), pp. 525–530.

[148] Rick Merritt. *Google designing AI processors*. 2016-05. URL: https://www.eetimes.com/google-designing-ai-processors/.

[149] Daniele Micciancio. “Generalized Compact Knapsacks, Cyclic Lattices, and Efficient One-Way Functions”. In: *Comput. Complex.* 16.4 (2007), pp. 365–411. DOI: 10.1007/s00037-007-0234-9. URL: https://doi.org/10.1007/s00037-007-0234-9.

[150] Daniele Micciancio and Petros Mol. *Pseudorandom Knapsacks and the Sample Complexity of LWE Search-to-Decision Reductions*. Cryptology ePrint Archive, Report 2011/521. https://eprint.iacr.org/2011/521. 2011.

[151] Gabrielle De Micheli and Nadia Heninger. “Recovering cryptographic keys from partial information, by example”. In: *IACR Cryptol. ePrint Arch.* (2020), p. 1506. URL: https://eprint.iacr.org/2020/1506.

[152] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. “Delphi: A Cryptographic Inference Service for Neural Networks”. In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/mishra.

[153] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. “DarkneTZ: towards model privacy at the edge using trusted execution environments”. In: *MobiSys ’20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*. 2020. DOI: 10.1145/3386901.3388946. URL: https://doi.org/10.1145/3386901.3388946.

[154] Fan Mo, Zahra Tarkhani, and Hamed Haddadi. “Machine Learning with Confidential Computing: A Systematization of Knowledge”. In: *ACM Comput. Surv.* 56.11 (2024), 281:1–281:40. DOI: 10.1145/3670007. URL: https://doi.org/10.1145/3670007.

[155] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX”. In: *Topics in Cryptology - CT-RSA 2018 - The Cryptographers’ Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*. 2018. DOI: 10.1007/978-3-319-76953-0_2. URL: https://doi.org/10.1007/978-3-319-76953-0%5C_2.

[156] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Cham, 2017. ISBN: 978-3-319-66787-4.

[157] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. “Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis”. In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020.

[158] Saja J. Mohammed and Dujan B. Taha. “Performance Evaluation of RSA, ElGamal, and Paillier Partial Homomorphic Encryption Algorithms”. In: *2022 International Conference on Computer Science and Software Engineering (CSASE)*. 2022. DOI: 10.1109/CSASE51777.2022.9759825.

[159] Payman Mohassel and Peter Rindal. “ABY^3: A Mixed Protocol Framework for Machine Learning”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018. DOI: 10.1145/3243734.3243760. URL: https://doi.org/10.1145/3243734.3243760.

[160] Payman Mohassel and Yupeng Zhang. “SecureML: A System for Scalable Privacy-Preserving Machine Learning”. In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 2017. DOI: 10.1109/SP.2017.12. URL: https://doi.org/10.1109/SP.2017.12.

[161] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang Nagel. “Detecting Memory-Boundedness with Hardware Performance Counters”. In: 2017-04. DOI: 10.1145/3030207.3030223.

[162] John V. Monaco. “SoK: Keylogging Side Channels”. In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 2018. DOI: 10.1109/SP.2018.00026. URL: https://doi.org/10.1109/SP.2018.00026.

[163] Koksal Mus, Yarkin Doröz, M. Caner Tol, Kristi Rahman, and Berk Sunar. “Jolt: Recovering TLS Signing Keys via Rowhammer Faults”. In: *IACR Cryptol. ePrint Arch.* (2022), p. 1669. URL: https://eprint.iacr.org/2022/1669.

[164] Koksal Mus, Saad Islam, and Berk Sunar. “QuantumHammer: A Practical Hybrid Attack on the LUOV Signature Scheme”. In: *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. 2020. DOI: 10.1145/3372297.3417272. URL: https://doi.org/10.1145/3372297.3417272.

[165] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. “New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*. 2019. DOI: 10.1007/978-3-030-22038-9_2. URL: https://doi.org/10.1007/978-3-030-22038-9%5C_2.

[166] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. “Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA”. In: *Information Systems* 92 (2020), p. 101524. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2020.101524.

[167] Krishna Giri Narra, Zhifeng Lin, Yongqin Wang, Keshav Balasubramanian, and Murali Annavaram. “Origami Inference: Private Inference Using Hardware Enclaves”. In: *14th IEEE International Conference on Cloud Computing, CLOUD 2021, Chicago, IL, USA, September 5-10, 2021*. 2021. DOI: 10.1109/CLOUD53861.2021.00021. URL: https://doi.org/10.1109/CLOUD53861.2021.00021.

[168] Deepika Natarajan, Andrew D. Loveless, Wei Dai, and Ronald G. Dreslinski. “Chex-Mix: Combining Homomorphic Encryption with Trusted Execution Environments for Oblivious Inference in the Cloud”. In: *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*. 2023. DOI: 10.1109/EUROSP57164.2023.00014. URL: https://doi.org/10.1109/EuroSP57164.2023.00014.

[169] Lucien K. L. Ng and Sherman S. M. Chow. “SoK: Cryptographic Neural-Network Computation”. In: *44th IEEE Symposium on Security and Privacy,*

SP 2023, San Francisco, CA, USA, May 21-25, 2023. 2023. DOI: 10.1109/SP46215.2023.10179483.

[170] Yue Niu, Ramy E. Ali, and Salman Avestimehr. "3LegRace: Privacy-Preserving DNN Training over TEEs and GPUs". In: *Proc. Priv. Enhancing Technol.* 2022.4 (2022), pp. 183–203. DOI: 10.56553/POPETS-2022-0105. URL: https://doi.org/10.56553/popets-2022-0105.

[171] Andrew M Odlyzko. "The rise and fall of knapsack cryptosystems". In: *Symposia of Applied Mathematics*. 1990.

[172] Keiji Omura and Keisuke Tanaka. "Density attack to the knapsack cryptosystems with enumerative source encoding". In: *IEICE transactions on fundamentals of electronics, communications and computer sciences* 87.6 (2004), pp. 1564–1569.

[173] 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: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. Denver, Colorado, USA, 2015. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813708. URL: http://doi.acm.org/10.1145/2810103.2813708.

[174] Claudio Orlandi, Alessandro Piva, and Mauro Barni. "Oblivious Neural Network Computing via Homomorphic Encryption". In: *EURASIP J. Inf. Secur.* 2007 (2007), pp. 1–11. DOI: 10.1155/2007/37343. URL: https://doi.org/10.1155/2007/37343.

[175] Todd Ouska. *Commit message: switch timing resistant exptmod to use temp for square instead of leaking key bit to cache monitor*. https://github.com/wolfSSL/wolfssl/commit/6ef9e79ff5ccd2b96fdfed404ada872fd29514be. 2016.

[176] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. "GPU Computing". In: *Proc. IEEE* 96.5 (2008), pp. 879–899. DOI: 10.1109/JPROC.2008.917757. URL: https://doi.org/10.1109/JPROC.2008.917757.

[177] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. "Practical Black-Box Attacks against Machine Learning". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United*

Arab Emirates, April 2-6, 2017. 2017. DOI: 10.1145/3052973.3053009. URL: https://doi.org/10.1145/3052973.3053009.

[178] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[179] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, 2016. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[180] Robert Philipp, Andreas Mladenow, Christine Strauss, and Alexander Völz. “Machine Learning as a Service: Challenges in Research and Applications”. In: *iiWAS ’20: The 22nd International Conference on Information Integration and Web-based Applications & Services, Virtual Event / Chiang Mai, Thailand, November 30 - December 2, 2020*. 2020. DOI: 10.1145/3428757.3429152. URL: https://doi.org/10.1145/3428757.3429152.

[181] David Pisinger. “Where are the hard knapsack problems?” In: *Comput. Oper. Res.* 32 (2005), pp. 2271–2284. DOI: 10.1016/j.cor.2004.03.002. URL: https://doi.org/10.1016/j.cor.2004.03.002.

[182] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. “ShowTime: Amplifying Arbitrary CPU Timing Side Channels”. In: *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*. 2023. DOI: 10.1145/3579856.3590332. URL: https://doi.org/10.1145/3579856.3590332.

[183] Hong Qin, Debiao He, Qi Feng, Muhammad Khurram Khan, Min Luo, and Kim-Kwang Raymond Choo. “Cryptographic Primitives in Privacy-Preserving Machine Learning: A Survey”. In: *IEEE Transactions on Knowledge and Data Engineering* (2023), pp. 1–17. DOI: 10.1109/TKDE.2023.3321803.

[184] Jan Reineke and Daniel Grund. “Relative competitive analysis of cache replacement policies”. In: *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*. 2008. DOI: 10.1145/1375657.1375665. URL: https://doi.org/10.1145/1375657.1375665.

[185] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. “Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications”. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*. 2018. DOI: 10.1145/3196494.3196522. URL: https://doi.org/10.1145/3196494.3196522.

[186] 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 Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. 2009. DOI: 10.1145/1653662.1653687. URL: http://doi.acm.org/10.1145/1653662.1653687.

[187] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems (Reprint)”. In: *Commun. ACM* 26.1 (1983), pp. 96–99. DOI: 10.1145/357980.358017. URL: https://doi.org/10.1145/357980.358017.

[188] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. “Deepsecure: scalable provably-secure deep learning”. In: *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. 2018. DOI: 10.1145/3195970.3196023. URL: https://doi.org/10.1145/3195970.3196023.

[189] Jonas Sander, Sebastian Berndt, Ida Bruhns, and Thomas Eisenbarth. “DASH: Accelerating Distributed Private Machine Learning Inference”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2025.1 (2025).

[190] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. “Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification”. In: *IEEE Access* 9 (2021), pp. 83067–83079. DOI: 10.1109/ACCESS.2021.3087421. URL: https://doi.org/10.1109/ACCESS.2021.3087421.

[191] Peter Scholl, Nigel P. Smart, and Tim Wood. “When It’s All Just Too Much: Outsourcing MPC-Preprocessing”. In: *Cryptography and Coding - 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings*. 2017. DOI: 10.1007/978-3-319-71045-7_4. URL: https://doi.org/10.1007/978-3-319-71045-7%5C_4.

[192] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. “Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs”. In: *CoRR* abs/1905.05725 (2019). arXiv: 1905.05725. URL: http://arxiv.org/abs/1905.05725.

[193] 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: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. Incheon, Republic of Korea, 2018. ISBN: 9781450355766. DOI: 10.1145/3196494.3196508. URL: https://doi.org/10.1145/3196494.3196508.

[194] 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*. Springer. 2017.

[195] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. “NetSpectre: Read Arbitrary Memory over Network”. In: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*. 2019. DOI: 10.1007/978-3-030-29959-0_14. URL: https://doi.org/10.1007/978-3-030-29959-0%5C_14.

[196] Mark Seaborn. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. 2015-03.

[197] Adi Shamir. “A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem”. In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. IEEE. 1982.

[198] Robert W. Shirey. *Internet Security Glossary, Version 2*. RFC 4949. 2007-08. DOI: 10.17487/RFC4949. URL: https://www.rfc-editor.org/info/rfc4949.

[199] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. "Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021-08. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman.

[200] Anatoly Shusterman, Zohar Avraham, Eliezer Croitoru, Yarden Haskal, Lachlan Kang, Dvir Levi, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. "Website Fingerprinting Through the Cache Occupancy Channel and its Real World Practicality". In: *IEEE Transactions on Dependable and Secure Computing* 18.5 (2021), pp. 2042–2060. DOI: 10.1109/TDSC.2020.2988369.

[201] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. "Util: : Lookup: Exploiting Key Decoding in Cryptographic Libraries". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2021. DOI: 10.1145/3460120.3484783. URL: https://doi.org/10.1145/3460120.3484783.

[202] Alessandra Silveira. "Automated individual decision-making and profiling [on case C-634/21-SCHUFA (Scoring)]". In: *UNIO–EU Law Journal* 8.2 (2023), pp. 74–85.

[203] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: http://arxiv.org/abs/1409.1556.

[204] Wei Song and Peng Liu. "Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. 2019. URL: https://www.usenix.org/conference/raid2019/presentation/song.

[205] Anna Cinzia Squicciarini, Cornelia Caragea, and Rahul Balakavi. "Toward Automated Online Photo Privacy". In: *ACM Trans. Web* 11.1 (2017), 2:1–2:29. DOI: 10.1145/2983644. URL: https://doi.org/10.1145/2983644.

[206] 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.

[207] Jakub Szefer and Ruby B. Lee. “A Case for Hardware Protection of Guest VMs from Compromised Hypervisors in Cloud Computing”. In: *31st IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2011 Workshops), 20-24 June 2011, Minneapolis, Minnesota, USA*. 2011. DOI: 10.1109/ICDCSW.2011.51. URL: https://doi.org/10.1109/ICDCSW.2011.51.

[208] Andrew S. Tanenbaum and Todd Austin. *Rechnerarchitektur*. Pearson, 2019. Chap. Chapter 3, pp. 1190–202.

[209] Harry Chandra Tanuwidjaja, Rakyong Choi, Seunggeun Baek, and Kwangjo Kim. “Privacy-Preserving Deep Learning on Machine Learning as a Service - a Comprehensive Survey”. In: *IEEE Access* 8 (2020), pp. 167425–167447. DOI: 10.1109/ACCESS.2020.3023084. URL: https://doi.org/10.1109/ACCESS.2020.3023084.

[210] Jan Philipp Thoma and Tim Güneysu. “Write Me and I’ll Tell You Secrets - Write-After-Write Effects On Intel CPUs”. In: *25th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2022, Limassol, Cyprus, October 26-28, 2022*. 2022. DOI: 10.1145/3545948.3545987. URL: https://doi.org/10.1145/3545948.3545987.

[211] M. Caner Tol, Saad Islam, Andrew J. Adiletta, Berk Sunar, and Ziming Zhang. “Don’t Knock! Rowhammer at the Backdoor of DNN Models”. In: *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*. 2023. DOI: 10.1109/DSN58367.2023.00023. URL: https://doi.org/10.1109/DSN58367.2023.00023.

[212] Florian Tramer. *slalom*. https://github.com/ftramer/slalom.git. 2019.

[213] Florian Tramer and Dan Boneh. “Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware”. In: *International Conference on Learning Representations*. 2018.

[214] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Stealing Machine Learning Models via Prediction APIs”. In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer.

[215] Anh-Tu Tran, The-Dung Luong, and Van-Nam Huynh. “A comprehensive survey and taxonomy on privacy-preserving deep learning”. In: *Neurocomputing* 576 (2024), p. 127345. DOI: 10.1016/J.NEUCOM.2024.127345. URL: https://doi.org/10.1016/j.neucom.2024.127345.

[216] Tomasz Tuzel, Mark P. Bridgman, Joshua Zepf, Tamas K. Lengyel, and Kyle J. Temkin. “Who watches the watcher? Detecting hypervisor introspection from unprivileged guests”. In: *Digit. Investig.* 26 Supplement (2018), S98–S106. DOI: 10.1016/J.DIIN.2018.04.015. URL: https://doi.org/10.1016/j.diin.2018.04.015.

[217] Jo Van Bulck, Frank Piessens, and Raoul Strackx. “Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018.

[218] 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: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016. DOI: 10.1145/2976749.2978406. URL: https://doi.org/10.1145/2976749.2978406.

[219] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. “GuardION: Practical Mitigation of DMA-based Rowhammer Attacks on ARM”. In: *Proceedings of the 15th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2018)*. 2018-06.

[220] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. “CacheQuery: learning replacement policies from hardware caches”. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*.

2020. DOI: 10.1145/3385412.3386008. URL: https://doi.org/10.1145/3385412.3386008.

[221] Pepe Vila, Boris Köpf, and José F. Morales. “Theory and Practice of Finding Eviction Sets”. In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 2019. DOI: 10.1109/SP.2019.00042.

[222] Michael Wang, Tingjun Yang, Maria Acosta Flechas, Philip C. Harris, Benjamin Hawks, Burt Holzman, Kyle Knoepfel, Jeffrey D. Krupa, Kevin Pedro, and Nhan Tran. “GPU-Accelerated Machine Learning Inference as a Service for Computing in Neutrino Experiments”. In: *Frontiers Big Data* 3 (2020), p. 604083. DOI: 10.3389/FDATA.2020.604083. URL: https://doi.org/10.3389/fdata.2020.604083.

[223] Qifan Wang, Lei Zhou, Jianli Bai, Yun Sing Koh, Shujie Cui, and Giovanni Russello. “*HT2ML*: An efficient hybrid framework for privacy-preserving Machine Learning using HE and TEE”. In: *Comput. Secur.* 135 (2023), p. 103509. DOI: 10.1016/J.COSE.2023.103509. URL: https://doi.org/10.1016/j.cose.2023.103509.

[224] Qizheng Wang, Wenping Ma, and Weiwei Wang. “B-LNN: Inference-time linear model for secure neural network inference”. In: *Inf. Sci.* 638 (2023), p. 118966. DOI: 10.1016/J.INS.2023.118966. URL: https://doi.org/10.1016/j.ins.2023.118966.

[225] Zhenghong Wang and Ruby B. Lee. “New Cache Designs for Thwarting Software Cache-based Side Channel Attacks”. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. San Diego, California, USA, 2007. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250723. URL: http://doi.acm.org/10.1145/1250662.1250723.

[226] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. “Piranha: A GPU Platform for Secure Computation”. In: *USENIX Security Symposium*. 2022.

[227] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. “DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD, 2018-08. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/weiser.

[228] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MicroWalk: A Framework for Finding Side Channels in Binaries”. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. San Juan, PR, USA, 2018. ISBN: 978-1-4503-6569-7. DOI: 10.1145/3274694.3274741. URL: http://doi.acm.org/10.1145/3274694.3274741.

[229] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. “Obelix: Mitigating Side-Channels Through Dynamic Obfuscation”. In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. 2024. DOI: 10.1109/SP54263.2024.00261. URL: https://doi.org/10.1109/SP54263.2024.00261.

[230] Rafal Wojtczuk. *TSX improves timing attacks against KASLR*. https://bromiumlabs.wordpress.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/. [Online; accessed 03-Nov-2020]. 2014.

[231] Henry Wong. *Intel Ivy Bridge Cache Replacement Policy*. 2013. URL: http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/.

[232] Xiaolong Wu, Aravind Kumar Machiry, Yung-Hsiang Lu, and Dave Jing Tian. “FlexGE: Towards Secure and Flexible Model Partition for Deep Neural Networks”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 22nd International Conference, DIMVA 2025, Graz, Austria, July 9-11, 2025, Proceedings, Part II*. 2025. DOI: 10.1007/978-3-031-97623-0_4. URL: https://doi.org/10.1007/978-3-031-97623-0%5C_4.

[233] 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.

[234] Gu Xiao-chen and Zhang Min-xuan. “Uniform Random Number Generator Using Leap Ahead LFSR Architecture”. In: *2009 International Conference on Computer and Communications Security*. 2009. DOI: 10.1109/ICCCS.2009.11.

[235] Andrew Chi-Chih Yao. “Protocols for Secure Computations (Extended Abstract)”. In: *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. 1982. DOI: 10.1109/SFCS.1982.38. URL: https://doi.org/10.1109/SFCS.1982.38.

[236] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[237] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “CacheBleed: a timing attack on OpenSSL constant-time RSA”. In: *Journal of Cryptographic Engineering* 7.2 (2017-06), pp. 99–112. ISSN: 2190-8516. DOI: 10.1007/s13389-017-0152-y. URL: https://doi.org/10.1007/s13389-017-0152-y.

[238] 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: *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE. 2009.

[239] Guangsheng Zhang, Bo Liu, Huan Tian, Tianqing Zhu, Ming Ding, and Wanlei Zhou. “How Does a Deep Learning Model Architecture Impact Its Privacy?” In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024 (in Print)*. 2024. URL: https://www.usenix.org/system/files/sec24summer-prepub-365-zhang-guangsheng.pdf.

[240] Jiliang Zhang, Congcong Chen, Jinhua Cui, and Keqin Li. “Timing Side-channel Attacks and Countermeasures in CPU Microarchitectures”. In: *ACM Comput. Surv.* 56.7 (2024), 178:1–178:40. DOI: 10.1145/3645109. URL: https://doi.org/10.1145/3645109.

[241] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. “CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds”. In: *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*. Cham: Springer International Publishing, 2016, pp. 118–140. ISBN: 978-3-319-45719-2. DOI: 10.1007/978-3-319-45719-2_6. URL: http://dx.doi.org/10.1007/978-3-319-45719-2%5C_6.