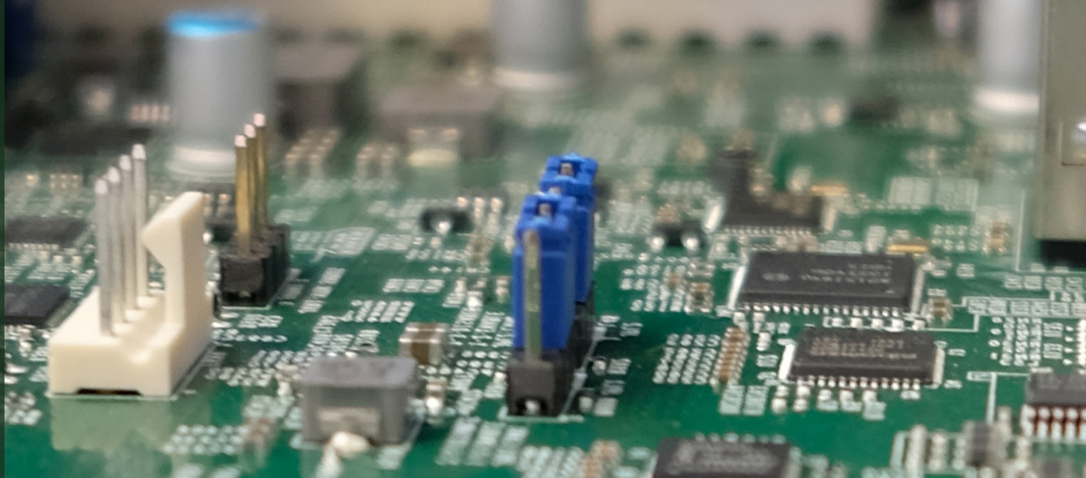


Understanding the Capabilities of Privileged Attackers Against Trusted Execution Environments

Ivan Puddu

Doctoral Dissertation ETH No. 29091

DISS. ETH NO. 29091

Understanding the Capabilities of Privileged Attackers Against Trusted Execution Environments

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

Ivan Puddu

MSc ETH CS, ETH Zurich

born on 18.08.1992

accepted on the recommendation of

Prof. Dr. Srdjan Čapkun, examiner
Dr.-Ing. Anil Kurmus, co-examiner
Dr. Jonathan M. McCune, co-examiner
Prof. Dr. Shweta Shinde, co-examiner

2023

Copyright © Ivan Puddu 2023
https://ivanpuddu.com

All rights reserved. Non-commercial use permitted.
http://rightsstatements.org/vocab/InC-NC/1.0/

DOI 10.3929/ethz-b-000610177

Contents

Abstract (English/Italiano/Français/Deutsch) vii

Abstract vii

Sommario ix

Résumé xi

Zusammenfassung xiii

Acknowledgements xvii

I Introduction and Background 1

1 Introduction 3
- 1.1 Thesis Contributions 6
- 1.2 Thesis Organization 9
- 1.3 Publications and Author Contributions 10

2 Background 13
- 2.1 Computer Architecture Fundamentals 13
- 2.2 Basic Isolation Enforcement 18
- 2.3 Trusted Execution Environments 27
- 2.4 Intel SGX 29
- 2.5 AMD SEV-SNP 36

II Information Leakage in TEEs 39

3 Frontal Attack 41
- 3.1 Introduction 41
- 3.2 Background 44
- 3.3 Overview of the Frontal Attack 46
- 3.4 Frontal Attack Profiling 49
- 3.5 Frontal Attack Exploitation 59
- 3.6 Affected Processors and Configurations 64

3.7 Potential Causes 65
3.8 Defenses 67
3.9 Related Work 68
3.10 Conclusions 72

4 Code Confidentiality in TEEs 75
4.1 Introduction 75
4.2 System and Attacker Model 77
4.3 Leakage Analysis Overview 80
4.4 Methodology 83
4.5 Leakage Analysis 85
4.6 IR Instruction Leakage in Practice 94
4.7 Evaluation 101
4.8 Related Work 104
4.9 Discussion 105
4.10 Conclusions 106

5 Preventing Single-Stepping 109
5.1 AEX-Notify 110

III Analysis of Attestation Shortcomings 113

6 Relay-safe Attestation 115
6.1 Introduction 115
6.2 SGX Background 119
6.3 Relay Attack Analysis 121
6.4 ProximiTEE 127
6.5 Experimental Evaluation 133
6.6 Addressing Emulation Attacks 140
6.7 Discussion and Related Work 145
6.8 Conclusions 146

7 Runtime Trust in Intel SGX 147
7.1 Introduction 147
7.2 Attacker Model: Manufacture-Time vs. Post-Deployment Compromise 148
7.3 Background 148
7.4 Special Cases for Message Recovery 153
7.5 Meet-in-the-Middle Attack on EGETKEY 154
7.6 Implications of the Attack 155

8 Runtime Trust in AMD SEV-SNP **157**
8.1 Introduction 157
8.2 Background 157
8.3 Attacks and Implications 157

9 Closing remarks **163**
9.1 Conclusions 163
9.2 Future work 164

Appendices **167**

A Frontal Attack **169**
A.1 Responsible Disclosure 169
A.2 Data-Oblivious Execution 169
A.3 Measurement Details 170
A.4 Outside Intel SGX 173

B Code Leakage **175**
B.1 Responsible Disclosure 175
B.2 x86 ISA Instruction Count 175
B.3 Analysis of SotA Attacker Cycle Accuracy 176

Bibliography **177**

List of Figures **189**

List of Tables **195**

List of Listings **197**

Abstract

Our lives today rely on the secure operation of computers in a diverse set of sectors, from energy to medicine. However, today's computers execute software bloated with complexity. Their large codebases provide a rich and versatile system, but most functionalities are often not needed in their target applications. This increases the trusted computing base (TCB) – the software and hardware that needs to be trusted for the system to work correctly. A large TCB is undesirable, as it gives attackers a higher likelihood to find and exploit vulnerabilities. Most of this complexity comes from the system software, that is, the operating system (OS) and the hypervisor. Despite this, the system software's codebase cannot generally be removed from the TCB, as it executes with the highest privileges.

Thanks to additional hardware primitives, Trusted Execution Environments (TEEs) break this paradigm, allowing even system software to be removed from the TCB. Most CPU manufacturers and architectures support some form of TEE: they can be found on Intel and AMD CPUs, as well as on ARM and RISC-V architectures. Their advent is promising, as they aim to let applications operate securely both when the (more privileged) system software is malicious and when a physical attacker can tamper with the system. Arguably, however, the guarantees that can be provided against such a strong and privileged attacker are not fully understood and often lead to TEE designs that make compromises invalidating the protections that they aim to provide. For example, previous work shows that the OS can abuse the CPU memory management interface to get notified when the TEE accesses attacker-specified memory regions, breaking data confidentiality. Understanding the capabilities of privileged attackers thus leads to more accurate designs and a more secure computing environment for everyone.

In this thesis, we contribute to the efforts of understanding the capabilities of privileged attackers in the context of TEEs in four main directions. First, we develop the *Frontal* attack, which shows that leveraging the OS to issue interrupts frequently leads to the CPU exposing detailed instruction execution timings, which can be used as a side channel. This side channel is detailed enough to leak encryption keys from a TEE and thus break data confidentiality. Second, we show that current commercial TEEs struggle to provide code confidentiality against a privileged attacker. Notably, we observe that using interpreters or JIT compilers inside TEEs – a popular choice due to their convenience and

flexibility – leaks significantly more confidential instructions compared to a baseline where native instructions are used instead.

The third and fourth contributions relate to attestation protocols, which are used to verify that a TEE is protecting a given application. We emphasize the impact of previously neglected aspects in attestation protocols in both these contributions. In the third contribution, we highlight that relay attacks, while once thought to be tolerable given the TEE protections, enhance the capabilities of a privileged attacker. Finally, in the fourth contribution, we show that current attestation protocols implicitly assume trust in the TEE manufacturer at runtime – despite the manufacturers often claiming otherwise. While this implicit trust in the TEE manufacturer is often overlooked, our analysis shows that it is a concrete threat in practice and should thus be accounted for in future attestation protocols.

Sommario

Oggigiorno, diversi aspetti della nostra vita dipendono dal corretto funzionamento dei computer in una vasta gamma di settori, dall'energia alla medicina. Tuttavia, attualmente i computer spesso eseguono software sovradimensionato e altamente complesso. La vastità di questo software fornisce un sistema versatile e ricco di funzionalità, anche se la maggior parte di queste spesso rimangono inutilizzate nelle applicazioni finali. Ciò aumenta la *Trusted Computing Base* (TCB, letteralmente *base fidata per la computazione*), ovvero il software e l'hardware di cui ci si deve fidare affinchè il sistema funzioni correttamente. Un'ampia TCB non è auspicabile, in quanto aumenta le probabilità che gli hacker trovino e sfruttino vulnerabilità per compromettere il sistema. La maggior parte di questa complessità deriva dal software di sistema, ovvero il sistema operativo e l'hypervisor. Nonostante ciò, la codebase del software di sistema non può essere generalmente rimossa dal TCB, poiché viene eseguita con i privilegi più elevati.

Grazie a delle primitive hardware aggiuntive, i *Trusted Execution Environments* (TEE, letteralmente *ambienti di esecuzione affidabili*) permettono di abbandonare questo paradigma, in quanto consentono di rimuovere dal TCB anche il software di sistema. La maggior parte dei produttori di CPU e delle architetture hardware supporta una forma di TEE: sono presenti sia sulle CPU di Intel e AMD, che sulle architetture ARM e RISC-V. Il loro avvento è promettente, in quanto mirano a proteggere applicazioni sia da un software di sistema (più privilegiato) malevolo, che da manomissioni fisiche sul sistema. Purtroppo, però, le garanzie che possono essere fornite contro un tipo di attacco così forte e privilegiato non sono ancora del tutto comprese e spesso portano a progettare TEE con compromessi che invalidano le protezioni che essi intendono fornire. Ad esempio, degli studi precedenti hanno dimostrato che il sistema operativo può abusare dell'interfaccia di gestione della memoria della CPU per essere notificato quando un TEE accede a regioni di memoria specificate dall'hacker, violando la riservatezza dei dati. La comprensione delle capacità di tipi di attacchi privilegiati porta quindi a progettare sistemi più accurati e a un ambiente informatico più sicuro per tutti.

In questa tesi, contribuiamo agli sforzi di comprensione delle capacità di tipi di attacchi privilegiati nel contesto dei TEE in quattro direzioni principali. In primo luogo, sviluppiamo l'attacco *Frontal*, che dimostra che sfruttando il sistema operativo per emettere interrupt frequentemente

porta la CPU ad esporre dettagliate tempistiche di esecuzione delle istruzioni, che possono essere utilizzate come canale laterale. Questo canale laterale è sufficientemente dettagliato da far trapelare le chiavi di crittografia da un TEE, così violando la sua riservatezza dei dati. In secondo luogo, dimostriamo che gli attuali TEE commerciali faticano a garantire la riservatezza del codice contro un attacco privilegiato. In particolare, osserviamo che l'uso di interpreti o compilatori JIT all'interno dei TEE – una scelta popolare per la loro convenienza e flessibilità – fa trapelare un numero significativamente maggiore di istruzioni confidenziali rispetto al caso in cui vengono invece utilizzate istruzioni native.

Il terzo e il quarto contributo riguardano i protocolli di attestazione, utilizzati per verificare che un TEE stia proteggendo una determinata applicazione. In entrambi questi contributi sottolineiamo l'impatto di aspetti precedentemente trascurati nei protocolli di attestazione. Nel terzo contributo, osserviamo che gli attacchi di tipo relay, un tempo ritenuti tollerabili visto le protezioni dei TEE, aumentano le capacità e pericolosità di un attacco privilegiato. Infine, nel quarto contributo, dimostriamo che gli attuali protocolli di attestazione presuppongono implicitamente la fiducia nel produttore dei TEE anche in fase di esecuzione, nonostante i produttori spesso affermino il contrario. Sebbene questa fiducia implicita nel produttore sia spesso trascurata, la nostra analisi mostra che si tratta di una minaccia concreta e che quindi dovrebbe essere tenuta in considerazione nei futuri protocolli di attestazione.

Résumé

Aujourd'hui, notre vie dépend du fonctionnement sécurisé des ordinateurs dans des secteurs très divers, de l'énergie à la médecine. Cependant, les ordinateurs d'aujourd'hui exécutent des logiciels gonflés de complexité. Leurs vastes bases de code fournissent un système riche et polyvalent, mais la plupart des fonctionnalités ne sont souvent pas nécessaires dans leurs applications cibles. Cela augmente la *base informatique fiable* (TCB) – les logiciels et le matériel qui doivent être fiables pour que le système fonctionne correctement. Une base de confiance importante n'est pas souhaitable, car elle ne fait que donner aux attaquants une plus grande probabilité de trouver et d'exploiter les vulnérabilités. La majeure partie de cette complexité provient du logiciel système, c'est-à-dire du système d'exploitation (OS) et de l'hyperviseur. Malgré cela, la base de code du logiciel système ne peut généralement pas être retirée de la TCB, car elle s'exécute avec les privilèges les plus élevés.

Grâce à des primitives matérielles supplémentaires, les *environnements d'exécution de confiance* (TEE) brisent ce paradigme, permettant même au logiciel système d'être retiré de la TCB. La plupart des fabricants et des architectures de processeurs supportent une certaine forme de TEE : on les trouve sur les CPU Intel et AMD et sur les architectures ARM et RISC-V. Leur avènement est prometteur, car ils visent à permettre aux applications de fonctionner en toute sécurité aussi bien lorsque le logiciel système (plus privilégié) est malveillant que lorsqu'un attaquant physique peut altérer le système. Cependant, les garanties qui peuvent être fournies contre un attaquant aussi puissant et privilégié ne sont pas entièrement comprises et conduisent souvent à des conceptions de TEE qui font des compromis invalidant les protections qu'elles visent à fournir. Par exemple, des travaux antérieurs montrent que le système d'exploitation peut abuser de l'interface de gestion de la mémoire du CPU pour être notifié lorsque le TEE accède à des régions de la mémoire spécifiées par l'attaquant, ce qui brise la confidentialité des données. Comprendre les capacités des attaquants privilégiés conduit donc à des conceptions plus précises et à un environnement informatique plus sûr pour tous.

Dans cette thèse, nous contribuons aux efforts de compréhension des capacités des attaquants privilégiés dans le contexte des TEEs dans quatre directions principales. Tout d'abord, nous développons l'attaque *Frontal*, qui montre que l'utilisation du système d'exploitation pour émettre des interruptions fréquentes conduit le CPU à exposer des temps d'exécution

d’instructions détaillés, qui peuvent être utilisés comme un canal latéral. Ce canal latéral est suffisamment détaillé pour faire fuir les clés de chiffrement d’un TEE et ainsi briser la confidentialité des données. Deuxièmement, nous montrons que les TEE commerciaux actuels ont du mal à assurer la confidentialité du code contre un attaquant privilégié. En particulier, nous observons que l’utilisation d’interprètes ou de compilateurs JIT dans les TEE – un choix populaire en raison de leur commodité et de leur flexibilité – entraîne la fuite de beaucoup plus d’instructions confidentielles par rapport à une base de référence où les instructions natives sont utilisées à la place.

Les troisième et quatrième contributions concernent les protocoles d’attestation, qui sont utilisés pour vérifier qu’un TEE protège une application donnée. Dans ces deux contributions, nous soulignons l’impact d’aspects précédemment négligés dans les protocoles d’attestation. Dans la troisième contribution, nous mettons en évidence le fait que les attaques de relais, qui étaient auparavant considérées comme tolérables compte tenu des protections du TEE, renforcent les capacités d’un attaquant privilégié. Enfin, dans la quatrième contribution, nous montrons que les protocoles d’attestation actuels supposent implicitement la confiance dans le fabricant du TEE au moment de l’exécution – bien que les fabricants prétendent souvent le contraire. Bien que cette confiance implicite dans le fabricant soit souvent négligée, notre analyse montre qu’elle constitue une menace concrète dans la pratique et devrait donc être prise en compte dans les futurs protocoles d’attestation.

Zusammenfassung

Unser heutiges Leben hängt vom sicheren Betrieb von Computern in den verschiedensten Bereichen ab, von der Energie bis zur Medizin mithilfe hochkomplexer Software. Ihre grossen Codebasen bieten ein reichhaltiges und vielseitiges System, aber die meisten Funktionen werden in ihren Zielanwendungen oft nicht benötigt. Dadurch erhöht sich die vertrauenswürdige Computerbasis (Trusted Computing Base – TCB) – die Software und Hardware, der vertraut werden muss, damit das System korrekt funktioniert. Eine grosse TCB ist unerwünscht, da sie nur eine höhere Angriffsfläche für Schwachstellen bietet. Der grösste Teil dieser Komplexität stammt von der Systemsoftware: dem Betriebssystem (OS) und dem Hypervisor. Trotzdem kann die Codebasis der Systemsoftware im Allgemeinen nicht aus der TCB entfernt werden, da sie mit den höchsten Privilegien ausgeführt wird.

Trusted Execution Environments (TEEs) versprechen dieses Paradigma zu ändern und ermöglichen es, Systemsoftware aus der TCB zu entfernen. Die meisten CPU-Hersteller und -Architekturen unterstützen irgendeine Form von TEE: Sie sind auf Intel- und AMD-CPUs sowie auf ARM- und RISC-V-Architekturen zu finden. Ihr Aufkommen ist vielversprechend, da sie darauf abzielen, Anwendungen sicher zu betreiben, sowohl wenn die (privilegiertere) Systemsoftware bösartig ist als auch wenn ein physischer Angreifer das System manipulieren kann. Die Garantien, die gegen einen so starken und privilegierten Angreifer gegeben werden können, sind jedoch nicht vollständig bekannt. Deshalb führen Designentscheidungen in TEE-Designs oft zu Kompromissen, die den angestrebten Schutz zunichtemachen. So haben frühere Arbeiten gezeigt, dass das Betriebssystem die Speicherverwaltungsschnittstelle der CPU missbrauchen kann, um benachrichtigt zu werden, wenn das TEE auf vom Angreifer spezifizierte Speicherbereiche zugreift, wodurch die Vertraulichkeit der Daten verletzt wird. In der Zukunft wird das Verständnis der Fähigkeiten privilegierter Angreifer somit zu genaueren Entwürfen und einer sichereren Computerumgebung für alle führen.

In dieser Arbeit leisten wir einen Beitrag zu den Bemühungen, die Fähigkeiten privilegierter Angreifer im Kontext von TEEs zu verstehen, und zwar in vier Hauptrichtungen. Erstens entwickeln wir den Frontal-Angriff, der zeigt, dass die Ausnutzung des Betriebssystems zur Ausgabe von Interrupts häufig dazu führt, dass die CPU detaillierte Zeitangaben zur Befehlsausführung preisgibt, die als Seitenkanal genutzt werden können.

Dieser Seitenkanal ist detailliert genug, um Verschlüsselungsschlüssel von einem TEE auszuspähen und so die Vertraulichkeit der Daten zu brechen. Zweitens zeigen wir, dass aktuelle kommerzielle TEEs kaum in der Lage sind, die Vertraulichkeit des Codes gegenüber einem privilegierten Angreifer zu gewährleisten. Insbesondere stellen wir fest, dass bei der Verwendung von Interpretern oder JIT-Compilern innerhalb von TEEs – eine beliebte Wahl aufgrund ihrer Bequemlichkeit und Flexibilität – deutlich mehr vertrauliche Anweisungen durchsickern lassen als bei der Verwendung nativer Anweisungen.

Der dritte und vierte Beitrag bezieht sich auf Bescheinigungsprotokolle, mit denen überprüft werden kann, ob ein TEE eine bestimmte Anwendung schützt. In diesen beiden Beiträgen betonen wir die Auswirkungen von bisher vernachlässigten Aspekten in Attestierungsprotokollen. Im dritten Beitrag heben wir hervor, dass Weiterleitungs-Angriffe, die früher angesichts des TEE-Schutzes als tolerierbar galten, die Möglichkeiten eines privilegierten Angreifers erweitern. Im vierten Beitrag schliesslich zeigen wir, dass die derzeitigen Bescheinigungsprotokolle implizit davon ausgehen, dass dem TEE-Hersteller zur Laufzeit vertraut wird – obwohl die Hersteller oft etwas anderes behaupten. Während dieses implizite Vertrauen in den Hersteller oft übersehen wird, zeigt unsere Analyse, dass es in der Praxis eine konkrete Bedrohung darstellt und daher in zukünftigen Attestierungsprotokollen berücksichtigt werden sollte.

Acknowledgements

It is curiosity that sparked my interest in pursuing a PhD. Curiosity to understand how a complex system like a computer works, its limits, and how they can be abused. Curiosity to see if I had what it takes to push the boundaries of knowledge – to be the first to figure out something that no one had before.

Curiosity, however, only goes so far. Reverse engineering undocumented CPU behavior is less enchanting than one idealizes it to be. And after you pour so much of yourself into a project, nothing really prepares you to deal with its paper being rejected again and again.

Nonetheless, despite the difficult times, I found my PhD journey to be incredibly rewarding. I owe that to the people I crossed paths with throughout these years, to those that stepped up when I needed them, and that helped me grow both scientifically and personally. Ultimately then, while curiosity was the spark, it was the people that were alongside me during this journey that kept the fire alive until the end. I hope the following words will do justice in acknowledging the impact each had in helping me see this dissertation to completion.

First, I would like to thank my thesis supervisor, Srdjan Čapkun. Thanks for giving me the opportunity to do a PhD, the freedom to explore different topics, and the safety to take risks in my research. Thanks for leading by example and showing me that excellence and results can be achieved while treating your students and postdocs with dignity and respect – something that sadly is far from being a given in academia – and, at times, even going above and beyond for them. Throughout these years, we shared many stimulating discussions about science, life, and, really, about everything. Through them, I got to know you on a personal level as well, and I feel fortunate to have found in you not just a mentor but also a great friend.

Thanks to my thesis committee, Srdjan, Jon, Anil, and Shweta, for volunteering some of their valuable time to be part of my thesis committee. Thanks also for all the detailed comments and the discussions we had about my thesis. The final version of this dissertation certainly benefitted from them. I am also grateful to Kari Kostiainen for his feedback on the writing of this dissertation, and to Ania Jomard and Moritz Schneider for their help with, respectively, the French and German translations of the thesis abstract.

My day-to-day work benefitted immensely from the interactions with the brilliant people in the System Security group at ETH: Aanjhan, Hubert,

Kari, Sinisa, Moritz, Aritra, Patrick, Mridula, Daniele, Karl, Marc, David, Friederike, Martin, Simon, Carolin, Claudio, and Giovanni, to name a few. Thanks for all the fun lunches together, trying to define what makes a sandwich a sandwich, a TEE a TEE, or figuring out who stole the pizzas from the fridge. I was always amazed at how easy it was to bounce technically challenging ideas off of you. Thanks also for the crazy memories from the conferences and summer schools that we traveled to together. I could not have asked for a better team to work with, and I am sure I will carry the friendships that formed with some of you throughout my whole life.

A particular mention goes to my office mate, Moritz Schneider. You impacted me and my research in many more ways than you probably realized. I always appreciated your way of arguing with me. Even when we had diverging views on a subject, you were able to quickly separate facts from speculation while staying respectful – allowing us to learn from each other's point of view. It fascinated me how often our discussions pondered on the meta and technicalities of topics seemingly unrelated to our work, all the while finding a way to influence and inspire it. I learned a great deal from interacting with you, not least the pragmatical but dedicated way you have to go about life and how you carry that into your work (and passions).

Thanks to my friends all over the world, and particularly in Zurich, Germany, the Netherlands, the UK, and Sardinia, for helping me keep my sanity and recharge my batteries while I worked on this thesis. Thanks for all the time we spent together, for the raclette dinners, for the ski trips, for the random trips across Europe, for always welcoming me no matter how long we were apart. Thanks for being there for me, not just to celebrate the victories but also to help me stand up again every time I fell.

It is hard to find the words to express how grateful I am to my better half, Ania, for her unwavering support throughout these years. Often the unsung heroine behind my work, making my life easy and light even through the stress of the looming deadlines. Thanks for having the patience to bear with "just until the next deadline" Ivan, and giving me the space I needed – I know how hard it was for you too. Thanks for showing me the pleasures that life has to offer and how to enjoy them to the fullest; that gave me the strength to keep pushing even when I thought I had none.

To my parents, Paride and Silvia: thanks for giving me the means to pursue my education, for the constant emotional support, and for enabling and empowering me to chase my dreams. You gave me the foundations to become the person I am today. To my siblings, Barbara and Massimo: thanks for always being there for me even when distance separates us. I know I will always be able to count on you no matter what. Thanks to my

family for always believing in me and what I can achieve. In you, I have my biggest supporters.

Finally, this thesis is dedicated to my grandfather, Ignazio Puddu. Thanks for instilling in me my machiavellian nature. Perhaps you did not make it to celebrate it with me, but you kept your promise anyway. I know how proud you would have been.

Ai miei genitori, Paride e Silvia: grazie per avermi dato i mezzi per sostenere i miei studi, per il costante supporto emotivo e per avermi permesso di inseguire i miei sogni. Devo a voi la persona che sono oggi. Ai miei fratelli, Barbara e Massimo: grazie per essermi sempre stati vicini nonostante la distanza ci separi. So che potrò sempre contare su di voi, qualunque cosa accada. Grazie alla mia famiglia per aver sempre creduto in me e in ciò che posso raggiungere. In voi ho i miei più grandi sostenitori.

Infine, questa tesi è dedicata a mio nonno, Ignazio Puddu. Grazie per aver instillato in me la mia natura macchiavellica. Forse non ce l'hai fatta a festeggiarla con me, ma so che hai mantenuto comunque la tua promessa. So quanto ne saresti stato orgoglioso.

Zurich, May 2023

Part I

Introduction and Background

Chapter 1
Introduction

The words of this thesis are far more likely to be read from the screen of a computing device than as ink etched on paper. The rapid evolution of computers in the past 70 years has led to their proliferation in every aspect of our lives. From the first bulky machines designed to aid the census and perform heavy computational workloads, computers have now become ubiquitous in our homes, offices, cars, airplanes, and play a crucial role in sectors ranging from banking to medicine.

Each of these different deployment scenarios has its own unique set of applications and security requirements. In a healthcare setting, for instance, a computer may be utilized to access patient health records and manage prescriptions. This computer may be running multiple applications such as a web browser, a patient record management application, and a PDF viewer. It is crucial that these applications are properly isolated to ensure the confidentiality and integrity of their data and code. Without this isolation, if a web browser is compromised, an attacker could access sensitive patient health records and tamper with prescriptions, with potential consequences to patients' lives. Similar considerations apply in virtually all the sectors where computers are employed.

Despite the potential harm, in today's computing environment, it is common for multiple applications to execute on the same computer or server. One reason for this is economic viability. Running multiple applications on the same hardware allows for better utilization of resources and reduces costs. This is particularly important in cloud computing, where servers are shared among multiple tenants. However, just as the different applications in the hospital example discussed above, these tenants should not have to trust each other for the correct functioning of their deployments.

Allowing mistrusting software to share the same hardware has been a concern since the early days of computing. Already in the 1960s, barely ten years since the advent of mainframes, researchers recognized the need for hardware and software cooperation to enable concurrent programs to execute securely. The seminal paper from Dennis and Van Horn [1] called for a supervisor software to manage processes with the aid of hardware primitives. These concepts stood the test of time, becoming the pillars that support isolation in today's computers. Today, two hardware mechanisms are particularly relevant in aiding isolation: *virtual addressing* and *hierarchical privilege levels*. The operating system (OS) and the hypervisor,

acting as the supervisor software, use paging to direct the central processing unit (CPU) virtual address (VA) translation. This gives each application and virtual machine (VM) the illusion of exclusive control of system memory. Hierarchical privilege level separation limits certain instructions and memory accesses to only be executed by the supervisor software, ensuring that low-privileged applications cannot modify VA mappings or access peripherals without cooperation from the OS.

The guarantees provided by these isolation mechanisms are well understood today. To hold, they rely on several assumptions, with the principal ones being the following two: i) the supervisor software is not compromised, and ii) no physical attacker, that is, the user and platform administrators are trusted. These assumptions are not exhaustive, as they, for instance, do not account for side channels [2], but they help in discussing the attacker model. If just one of these assumptions is broken, then the whole system is compromised. The first assumption is deeply linked to the hierarchical privilege layers and to the concept of the trusted computing base (TCB) – the software and hardware components that need to be trusted for the security of the overall system to be guaranteed. Due to it being more privileged, supervisor software is part of the TCB of every VM and application running in the system. Given its part in the TCB of every application, it is therefore crucial for supervisor software to not have any vulnerability.

The supervisor software does not just isolate applications from each other: it executes a scheduler, drivers, the filesystem, and manages network connections, to name a few. All these capabilities come at the cost of code size and complexity, which widens the attack surface. In short, having a large monolithic OS control the entire system makes it difficult to guarantee or verify a compromise-free environment. Thus, the damage inflicted by malware cannot be fully contained in this attacker model. Even if malware is installed as an unprivileged application, once it escalates its privileges to the privileges of supervisor software, it can control the entire system.

The second assumption, the absence of a physical attacker, is also problematic to justify in modern systems. Particularly, in a cloud computing setting, the cloud provider might not be fully trusted. An untrustworthy cloud provider has several avenues to compromise the execution of one of its tenants. Two exemplary ones are cold boot attacks [3] and employing a malicious hypervisor. Cold boot attacks allow to extract the content of DRAM. Even if the OS protects software accesses to peripherals, a physical attacker can completely bypass these restrictions. Having control of the hypervisor has similar consequences to

compromising the OS: the hypervisor is part of the TCB of every VM, so it can read or tamper with the tenants' VMs memory. The difference with the first assumption is that the malicious cloud provider does not need to find a vulnerability in the hypervisor – they can just install a malicious one to target a specific user. In summary, not only do both assumptions weakly hold in modern systems, but they are also impossible to verify by the user.

Trusted Execution Environments. The increasing difficulty in meeting these assumptions led in the 2000s to the first proposals aiming to remove the supervisor software from the TCB of applications and VMs [4, 5, 6, 7, 8]. These proposals and the many that followed with the same goals [9] are commonly known today as trusted execution environments (TEEs). While TEEs are still an area of active research, recently, they started being rolled out in major architectures. Some examples are Intel SGX [10], AMD SEV-SNP [11], ARM TrustZone [8], ARM CCA [12], and even on RISC-V with Keystone [13].

TEEs allow isolating *enclaves*[1] from a malicious environment by leveraging hardware primitives. Intel SGX and AMD SEV are the two most prevalent commercial TEE approaches on general-purpose devices. They aim to protect both against compromised supervisor software and against a physical attacker. Protection against a physical attacker is achieved thanks to the CPU encrypting and authenticating the enclave memory when stored in DRAM. Thus, even with a cold boot attack, at best, only encrypted memory can be recovered from DRAM.

Doing away with the supervisor software in the TCB requires two hardware-supported primitives: *tracking and isolating* the enclave context and *attestation*. Intel SGX and AMD SEV track context switches in and out of the enclave and ensure that only code running inside the enclave can access its memory and resources. The difference between the two approaches in this regard is that SGX enclaves include only low-privileged applications, while SEV enclaves isolate a full VM. Finally, attestation ties all of these protections together. The software itself cannot ascertain whether it is executing within an enclave or as a regular application because the OS could emulate the enclave environment. Attestation solves this impasse in TEEs by allowing to prove to a remote verifier, e.g., a client of a cloud service, that a remote code has been deployed inside an enclave. Attestation is based on cryptographic methods, wherein the CPU computes

[1] Different implementations refer to the environment protected by the TEE in different ways. For instance, ARM CCA refers to them as *realms*. Throughout the thesis, we always refer to the environment protected by a TEE as *enclave*, irrespective of how the particular implementation refers to it.

and signs the identity of the enclave with a secret key. Since the supervisor software cannot access the CPU secret keys, the remote verifier can securely deploy secrets to enclaves upon successful attestation, as it proves that the TEE protections are in place.

Protecting against a strong attacker, despite the hardware protections, is more challenging than initially thought. Enclaves still need to communicate with the system software through system or supervisor calls. The results of these calls can, however, be modified by the attacker through what is known as an Iago attack [14]. Control over the supervisor software allows noise reduction for traditional side-channel attacks [15, 16]. The capability to inspect and modify page tables allows controlled-channel attacks [17, 18, 19], a side channel entirely controlled by the attacker. Particularly, with controlled-channel attacks, the attacker can modify page tables so that the CPU notifies the attacker when the enclave accesses certain parts of its memory [17]. The attacks can also be made more stealthy by simply monitoring the accessed and dirty bits of the page tables [18, 19]. Having physical access and supervisor capabilities, the attacker can also undervolt the CPU [20, 21], glitching the execution in enclave mode and thus tampering with its integrity.

At their core, hierarchical isolation primitives and TEEs rely on the same principle: isolating memory between execution contexts. However, isolation against privileged attackers must account for all of the resources and the control they can exert over the system, as the described attacks highlight. While we understand how to effectively isolate against an unprivileged attacker, our understanding of the capabilities of a privileged attacker is not yet sufficient to fully meet the guarantees that TEEs promise.

1.1 Thesis Contributions

Increasing the understanding of the capabilities of a privileged attacker guides the design of the software and hardware of TEE stacks. In this thesis, we contribute to the understanding of the capabilities of the attacker in four main directions. The first two focus on the impact that the ability to control interrupts has on instruction timing and confidential code leakage, while the last two highlight the need for attestation protocols to provide additional properties. In the following, we discuss these contributions in more detail.

Timing Leakage Under Frequent Interrupts. We investigate the impact that issuing frequent interrupts has on instructions' timings. Our evaluation and experiments show that, when frequently issuing interrupts, instruction

execution times correlate with their virtual address. Particularly, we show that the feature of the virtual address that explains the timing variability is its alignment with respect to the CPU fetch window – the range of memory that gets fetched by the CPU to be decoded as instructions depending on the current program counter. We show that this correlation emerges when frequently issuing interrupts but not during normal execution – when the pipeline is not frequently flushed. Thus this attack can only be leveraged by a privileged attacker, e.g., in the context of a TEE. We also observe that since the correlation occurs based on the address, even the execution of the same instruction can produce different timings.

We leverage these observations to introduce the *Frontal* attack against Intel SGX enclaves. The Frontal attack leaks fine-grained control flow in branches containing the same instructions, even when they span less than a cacheline in size – which has been proven challenging in previous work. We demonstrate that it can achieve an accuracy of 99% in our synthetic binaries, but the resolution on real binaries is usually less. Nonetheless, we use it to exploit two cryptographic libraries: the Intel IPP Cryptography library and the mbedTLS library. For the mbedTLS proof-of-concept, we perform an end-to-end attack, using the Frontal attack to leak a full RSA key.

Evaluation of Code Leakage in TEEs. TEEs aim to provide not only data confidentiality but also code confidentiality. Several commercial [22, 23, 24, 25] and academic [26, 27, 28, 29, 30, 31, 32, 33] solutions exist aiming at leveraging code confidentiality in TEEs. We study whether code confidentiality can be truly provided on current TEEs, given the capabilities of privileged attackers. We generalize the existing proposals and show that they generally follow two approaches: native execution and intermediate representation (IR) execution. In native execution, the enclave executes the confidential instructions natively, while in IR execution, an interpreter or just-in-time compiler inside the enclave executes the confidential instructions. We develop a methodology to quantify instruction leakage under different privileged attacker strengths. The methodology leverages the fact that certain types of instructions leave different traces in the system, which can then be combined to construct a list of *candidate instructions*. We apply our methodology to both Intel and AMD CPUs. Our results highlight that IR execution inherently presents an amplification leakage compared to native execution because to execute one IR execution, multiple native instructions are required – the leakage of which can be combined.

While in native execution, less than 10 % of the ISA instructions can be leaked, the observed leakage amplification in IR execution is significant. To showcase the danger with IR execution, we experimentally demonstrate a practical end-to-end instruction extraction attack against WAMR, a WASM runtime running on Intel SGX. Our attack was able to fully classify known programs and subroutines running inside the enclave and was able to extract with 100% confidence around 50% of an enclave executing a previously unseen program.

Relay-Safe Attestation. While relay attacks have been known for over a decade, their implications have not been thoroughly analyzed. TEEs' attestation protocols do not usually aim at preventing relay attacks. The rationale behind this choice is that attestation proves to the verifier that it is interacting with a genuine CPU. Since the CPU is genuine, it should not matter in which computer it is exactly. We show, however, that giving the attacker the capability to relay attestation to another platform enhances its capabilities. We observe that relay attacks increase the capability of the attacker to leverage side-channel attacks. For instance, they allow an adversary with only remote access to a platform (say after a compromise) to relay the attestation to a local platform over which it can also perform physical attacks (such as voltage manipulation). Relay attacks also allow a form of "downgrade attack" in which the attacker can relay to a platform that will never receive security updates and thus be exploitable in the future.

To mitigate these attestation relay attacks, we propose ProximiTEE, which enhances attestation with proof of proximity to a particular CPU. ProximiTEE relies on a physical device that can be connected to the specific target platform that the user wants to attest to. ProximiTEE requires no changes from the manufactures hardware and attestation designs at the cost of the extra device, which has to be connected to the server. We evaluate ProximiTEE and show with a prototype that relay prevention is practical in practice.

Security Analysis of Manufacturer's Runtime Trust. Trust in the hardware manufacturer is implicitly assumed when working with TEEs. Usually, the argument is that since the manufacturer produces the hardware that is being adopted, if they were to be malicious, they would simply compromise the hardware. This often leads to tolerating them as active parties in TEEs attestation protocols. We observe that manufacturers might not be malicious at manufacturing time or that their manufacturing facilities might not be compromised, yet their attestation facilities could be compromised, or they

could be compelled (e.g., with a lawful order) to compromise them. We thus argue that there is an advantage in separating the trust in manufacturers between manufacturing-time trust and runtime or post-deployment trust.

Intel and AMD claim that they cannot compromise attestation post-deployment. However, through our security analysis of their protocols and public patents, we show that if they were to be compromised at runtime, their current attestation mechanisms could be abused by an attacker (particularly a privileged one). With this analysis, we highlight the need to design attestation protocols that do not require runtime trust in the manufacturer.

1.2 Thesis Organization

This thesis is organized into three main parts, Parts I to III. Part I opens up the thesis with the Introduction (this chapter) and the background in Chapter 2. Chapter 2 discusses the background relevant to the rest of the thesis, particularly focusing on Intel SGX and AMD SEV primitives. The rest of the thesis consists of Part II, related to information leakage in TEEs, and Part III, on analyzing attestation protocols shortcomings. In Part II, we discuss data leakage with the Frontal attack in Chapter 3 and introduce our methodology and analysis of code leakage in Chapter 4. The results of both Chapters 3 and 4 rely on the attacker's capability to fine-tune the delivery times of interrupts. We thus conclude Part II with Chapter 5 by discussing AEX-Notify, a new architectural extension with which Intel aims to curb these attacks.

In Part III, we discuss the dangers of relay attacks in Chapter 6, where we also discuss an enhancement to existing attestation protocols to prevent these attacks. Chapters 7 and 8 then conclude Part III by discussing the nuances and importance of separating the trust in the manufacturer between manufacturing time trust and deploy-time trust. The analysis in these chapters focuses mainly on how Intel (in Chapter 7) and AMD (Chapter 8) need to be trusted at runtime in their current attestation protocols.

Finally, we discuss the closing remarks of this thesis and future work in Chapter 9.

1.3 Publications and Author Contributions

This thesis is based, in part, on the following publications:

[P1] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun. "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend". *30th USENIX Security Symposium (USENIX Security '21)*.

[P2] Ivan Puddu, Moritz Schneider, Daniele Lain, Stefano Boschetto, and Srdjan Čapkun. "On (the Lack of) Code Confidentiality in Trusted Execution Environments". *arXiv preprint 2212.07899 (2022) - Under submission*.

[P3] Aritra Dhar, Ivan Puddu, Kari Kostiainen, and Srdjan Capkun. "ProximiTEE: Hardened SGX Attestation by Proximity Verification". *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY '20)*. **Best Paper Award**.

These publications are the result of the collaboration and contributions of all the involved authors. In the following, when necessary, we refer to the authors of these publications with their initials. As Chapters 3, 4 and 6 are respectively based on [P1, P2, P3], they also contain contributions from the coauthors of the respective papers, including on the writing and figures. The same applies to Chapter 7, which is based on an unpublished report, coauthored with *SC*, and first written to coordinate a responsible disclosure. Next, we summarize the contributions made by this thesis' author, *IP*, to these four chapters.

Author Contributions. For Chapter 3, derived from [P1], IP implemented the code of the experiments, the proof-of-concept, and the defenses. In cooperation with the other coauthors, IP designed, executed, and analyzed the data of the experiments to reverse engineer the CPU behavior. IP performed the binary and source code analysis of the Intel IPP and mbedTLS library and executed and evaluated the attacks on the found vulnerabilities. IP evaluated the effectiveness of the defenses. IP coordinated the project across collaborators. All the authors contributed to the final version of the published manuscript.

For Chapter 4, derived from [P2], IP implemented the code of the framework to compute the candidate sets and collected the data. In cooperation with the other coauthors, IP analyzed and interpreted the instruction leakage data for native and IR execution. IP executed the experiments to collect the profiling data for the WASM segmentation

attacks. IP designed, implemented, and evaluated the end-to-end WASM attack. IP coordinated the project across collaborators. All the authors contributed to the final version of the published manuscript.

For Chapter 6, derived from [P3], IP, in cooperation with the other authors, conceptualized the risks posed by relay attacks, designed the protocols, and performed the security analysis. IP and *AD* implemented the protocols and collected the experimental data. In cooperation with the other authors, IP analyzed the data. All the authors contributed to the final version of the published manuscript.

For Chapter 7, derived from an unpublished report authored by IP and SC, IP had the original idea and conceptualized the attack. IP and SC defined the concrete threats posed by the attacks. All the authors contributed to writing the report.

1.3.1 Other Publications

During his doctoral studies, the author of this thesis also coauthored the following publications, which are closely related to the work of this thesis:

[R1] Ivan Puddu, Daniele Lain, Moritz Schneider, Elizaveta Tretiakova, Sinisa Matetic, and Srdjan Capkun. "TEEvil: Identity Lease via Trusted Execution Environments". *arXiv preprint 1903.00449 (2019)*.

[R2] Jawad Haj-Yahya, Jeremie S. Kim, A. Giray Yağlıkçı, Ivan Puddu, Lois Orosa, Juan Gómez Luna, Mohammed Alser, and Onur Mutlu. "IChannels: Exploiting Current Management Mechanisms to Create Covert Channels in Modern Processors". *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21)*.

[R3] Moritz Schneider, Aritra Dhar, Ivan Puddu, Kari Kostiainen, and Srdjan Čapkun. "Composite Enclaves: Towards Disaggregated Trusted Execution". *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021).

[R4] Friederike Groschupp, Mark Kuhne, Moritz Schneider, Ivan Puddu, Shweta Shinde, and Srdjan Capkun. "It's TEEtime: Bringing User Sovereignty to Smartphones". *arXiv preprint 2211.05206 (2022) - Under submission*.

In addition, the author of this thesis worked on projects that led to the following publications:

[O1] Patrick Leu, Ivan Puddu, Aanjhan Ranganathan, and Srdjan Capkun. “I Send, Therefore I Leak: Information Leakage in Low-Power Wide Area Networks”. *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec ’18)*.

[O2] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. “Project PBerry: FPGA Acceleration for Remote Memory”. *HotOS ’19*.

[O3] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. “Rethinking Software Runtimes for Disaggregated Memory”. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’21)*.

[O4] Lois Orosa, Yaohua Wang, Mohammad Sadrosadati, Jeremie S. Kim, Minesh Patel, Ivan Puddu, Haocong Luo, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Nika Mansouri-Ghiasi, Saugata Ghose, and Onur Mutlu. “CODIC: A Low-Cost Substrate for Enabling Custom In-DRAM Functionalities and Optimizations”. *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA ’21)*.

Chapter 2
Background

This chapter introduces the relevant background needed to understand the rest of the thesis. We first describe basic computer architecture topics, particularly showing how context switches work in a hierarchical privilege-level system. We then explain what changes trusted execution environments (TEEs) make to these basic systems. We generally focus on Intel SGX, AMD SEV-SNP, and ARM CCA, as at their core, these try to achieve similar objectives and are the most widely deployed TEEs to date. We then describe in more detail how SGX enclaves are initialized and how the operating system manages enclave context switches. Finally, we discuss how these mechanisms translate to AMD SEV enclaves.

2.1 Computer Architecture Fundamentals

We now give a basic overview of a modern *computer architecture*. Computer architecture concerns the design of a computing system at different levels of abstraction: From how it is subdivided into individual hardware components, to the internal implementation of each of these components, and how they interface with each other and to software. Throughout this section, we touch on different design aspects of a modern computer architecture, focusing on how they give rise to security concerns and what mechanisms have been introduced to address these concerns.

2.1.1 Hardware and Software

One of the crucial design aspects of a modern computing system is whether a given functionality should be implemented in hardware or software. This choice has profound implications in terms of performance, flexibility, cost, and security.

In terms of performance, generally, implementing a functionality on a dedicated hardware component or module makes it execute more efficiently and faster than if it was implemented by software. The downsides come in flexibility. As these components are tailor made for their target functionality, they could be under-utilized if their functionality is not often needed. On the other hand, general-purpose hardware, which is fully customizable by software, has the benefit of supporting virtually any application, with a trade-off in efficiency and performance compared to dedicated hardware.

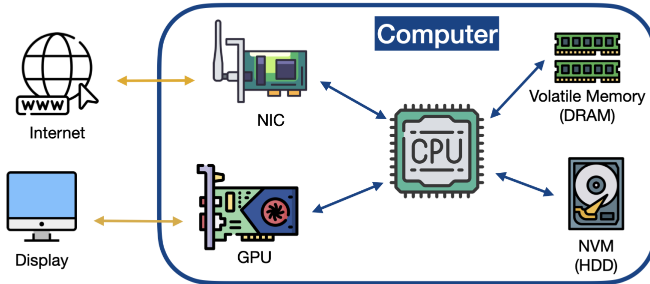


Figure 2.1: *Example of a typical hardware configuration on a modern computing architecture. The edges between the components show the communication topology. Blue (dark) edges represent typical internal communication links, while the yellow (light) ones show how external entities are connected.*

The most ubiquitous architecture nowadays, a CPU-centric architecture, strikes a balance at a system's level between these two extremes. We give a simplified view of this architecture in Figure 2.1. The figure depicts the various components found in the architecture and how they communicate with each other and the external world. At the core of the architecture, we can find the central processing unit (CPU), the only general-purpose hardware component in the system. The remaining hardware components are usually tasked with data acquisition, storage, and communication, to name a few. With this configuration, software that specializes the system for a particular application is mostly meant to execute on the CPU. This leaves other components with the job of aiding the CPU in executing specific sub-tasks. For example, the network interface card (NIC) and the graphics processing unit (GPU), depicted in the figure, help the CPU in networking and graphics processing tasks, respectively. While the CPU could perform both of these tasks, the NIC and the GPU do it far more efficiently than the CPU could, all the while relieving the CPU from performing them, thus freeing its computational resources for other tasks. Note how these two tasks are needed by a wide range of applications, thus justifying the presence of dedicated hardware for them despite their narrow functionalities. A second type of peripherals, shown in the picture, provides custom hardware for various types of memory extension. The CPU itself has some memory, such as read-only memory (ROM), which is non-volatile, and registers and caches, which are volatile[1]. What is important to take away from this is that

[1]Volatile memory is memory that is not guaranteed to be preserved upon a system reboot.

the CPU has a limited amount of memory, generally not enough to handle the requirements of today's applications. This is why we find on the system separate hardware components providing additional pools of memory. That said, we will not delve into the advantages and disadvantages of having these non-integrated in the CPU, as it is out of scope for the thesis.

This brief description of the architecture is sufficient to give an intuition as to how these different components work together to form a computing system capable of handling a wide spectrum of applications, from weather forecasting to playing cute cat videos. Intuitively, each component could benefit from directly communicating with all the others. For instance, if a network packet just contains a frame that needs to be rendered on screen, the NIC could instruct the GPU to process it. However, direct communication between each component would go against the design principle we saw before, wherein each hardware component (besides the CPU) is specialized to perform only its given task. In the example of the video frame, the NIC would need to know how to interface itself with the particular GPU installed in the system. The problem with this approach is that the interface of these components is vastly different, not only across different manufacturers but also between models and versions. Recall that their advantage is to have functionality implemented in hardware, and having to support every other possible device in hardware immensely raises the complexity and cost of the hardware design. As a consequence, manufacturers release a particular type of software, known as *driver*, to facilitate this communication. Drivers expose an application programming interface (API) to the rest of the system, which allows other software to interact with the device. As drivers are software, they execute on the only component in the system designed to handle it: the CPU. Because all drivers run in the CPU, the CPU can communicate with all devices. Therefore, for the example mentioned above, the frame received by the NIC is first sent to the CPU, which can talk to the NIC thanks to the specific NIC's driver executing on the CPU. An application's software that received the network packet, and also executing on the CPU, then sends the frame to the GPU, thanks to the GPU drivers installed on the system.

To summarize, due to the fact that drivers execute on the CPU, the CPU is able to communicate with every component on the system, and thus takes in the role of setting up every component, and orchestrating the communication between them. This is shown by the blue (dark) edges in Figure 2.1, which show which communication links form between the components on the platform. The figure highlights how a star topology emerges wherein the CPU is involved in the communication between all

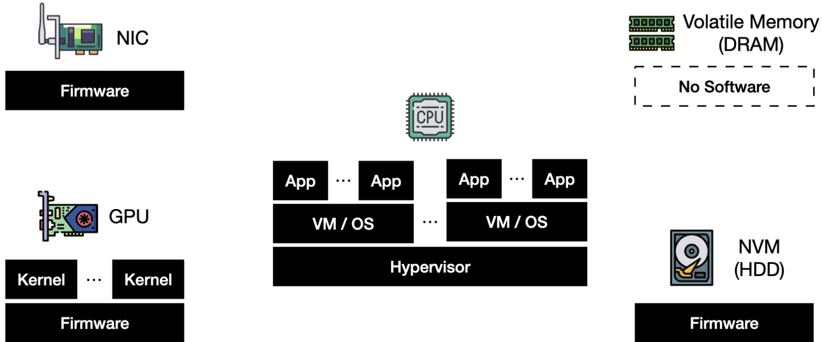


Figure 2.2: *Software organization on the system*

the other components. The only exception to this communication topology occurs with direct memory access (DMA). With DMA, the CPU can configure a portion of system memory (DRAM) to be available to any other peripheral in the system that supports it. The peripheral assigned to a particular range can then read and write to it without involvement from the CPU. This and the rest of the memory management has security implications, as we will see in the next sections.

2.1.2 The Trusted Computing Base

We depict the software executing on the various components of the system in Figure 2.2. Not surprisingly, as discussed before, most of the software executes in the CPU. This is indicated in the figure by the various applications, the Operating System (OS), Virtual Machines (VMs), and hypervisor executing on the CPU. Nonetheless, some software also executes on the other hardware components present in the system. For the most part, this software is known as *firmware*, and it takes care of managing several internal implementation-specific hardware resources and provides an interface to the drivers running on the CPU.

This type of software distribution across the system is typical in a CPU-centric architecture. The fact that the system can support multiple applications, even running simultaneously, is a testament to the versatility of this configuration. However, utilizing the same hardware for multiple applications comes with security risks. To explain them and how they are addressed, we briefly comment on how the system can be abused if no protection is in place. Imagine a scenario in which a banking application is executing on the system alongside another utility application, say a music

player. Both applications execute on the CPU, and thus have access to system memory and all of the peripherals. Imagine that the video player can modify the memory of the banking app by, say, sending requests to DRAM. No integrity can be guaranteed in this case, and a user of this system needs to ensure that no application is malicious in order to properly utilize any of them. This becomes quickly infeasible the more applications are installed on the system. Besides this, drivers and firmware executing on the system also have access to system memory and are thus capable of compromising any other app on the system if they are malicious.

Informally, the set of software and hardware that needs to be trusted for a particular task to execute as intended is known as the *trusted computing base* (TCB). Without any protection, the whole system is part of the TCB, meaning that no software or hardware should be malicious for the system to operate correctly. With a system-wide TCB, as soon as a malicious application is installed in the system, then that application could steal every other application's data or even modify and delete the code and data of every other application installed. With a TCB this large, the question is not how to protect a system but how long it will take until it gets compromised. This ultimately would erode users' trust in the whole system, drastically reducing the set of applications for which a computer could even be considered a viable option.

Modern platforms tackle this problem by trying to minimize the TCB needed to execute an application. We refer to the smallest set of hardware components and software needed to execute an application as its *security context*. Independent applications can be engineered to have separate security contexts, as they do not need to rely on each other to execute. Ideally, these independent security contexts executing on the system would then have a separate TCB. Modern platforms try to *isolate* different security contexts on the system through several mechanisms, which we introduce in the next sections. Isolation between security contexts commonly refers to memory isolation between them. Informally, if two contexts are isolated, they are not able to read or write each other's memory. The idea behind this is that the security measures between the two contexts should achieve the same level of isolation as the ideal world in which each security context is executing on its own separate platform. Next, we see how this isolation is enforced in practice.

2.2 Basic Isolation Enforcement

Observe that if a hardware component only has a security context, then no isolation mechanisms are needed in that component. Take the software running on the NIC, shown in the top left corner of Figure 2.2. Since the NIC only needs to execute its own firmware, there is no need to separate the TCB *within* it. However, for a transaction to be sent correctly, a banking app should not have to trust any of the productivity software or video games installed in the system to be benign. As discussed, due to the way the platform is organized, these mutually distrusting software components execute on the CPU, and thus naturally, most of the TCB separation mechanisms are implemented and enforced on the CPU. Note, however, that other peripherals, such as the GPU [34], also implement similar principles when they need to separate their security contexts.

Isolation between different security contexts can be achieved with mechanisms that range from pure software to pure hardware. Pure software mechanisms work by injecting instructions into untrusted instruction streams. These injected instructions enforce restrictions for untrusted processes so that, e.g., they can only access specific memory regions. With hardware mechanisms, instead of injecting/modifying instructions, the hardware itself enforces the restrictions.

Software Fault Isolation (SFI) [35], interpreters, and just-in-time (JIT) compilers are examples of pure software mechanisms that can be used to achieve isolation. These mechanisms are also often referred to as *sandboxing*. Practically, however, isolation enforcement is often realized by a cooperation between hardware and software, and thus we focus on these mechanisms.

Several hardware mechanisms have been devised over the years to create TCBs compartmentalizing different modules of the software and hardware stack. The primitives themselves define how tight the TCB can be made around a given security context. In line with the principle of least privilege, the tighter the TCB is around the security context, the better it is in terms of security, as this minimizes the attack surface as much as possible. In the following, we introduce different primitives used to isolate the various security context running on the system. Particularly how software is compartmentalized into privilege levels and how virtual memory management aids in achieving a finer separation within each level. These two primitives work together to create separate TCBs on the system, and understanding them and their limitations is key to understanding what more modern solutions are aiming to achieve.

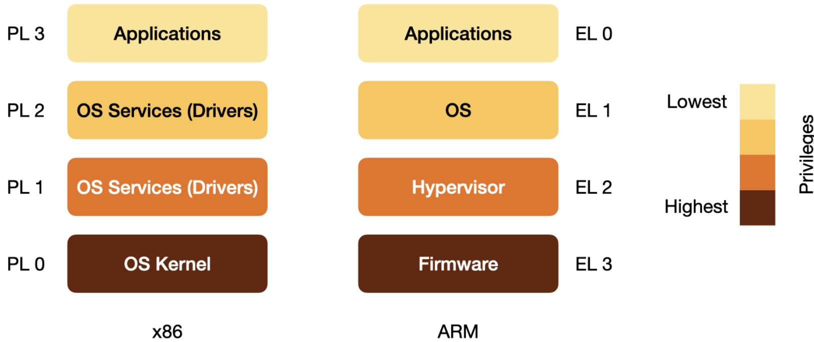


Figure 2.3: *Privilege levels and expected software to be running on each level (according to [37, 39]) on x86 (left) and ARM (right). Note that numbers decrease (increase) with more privileges in x86 (ARM).*

2.2.1 Privilege Levels

The first principle that we discuss is that of hierarchical execution contexts. With this principle, permissions and capabilities are based on the current *execution context*, which we define next. The execution context generally defines the current state of the CPU. Register values, internal signals, and the current memory state are some of the elements that determine the CPU's execution context. The CPU can provide different capabilities to the executing instructions, depending on the current execution context. The allowed capabilities are often organized in hierarchical levels [2], with more privileged levels inheriting and extending all the capabilities from the lower-privileged levels. This principle is implemented in practically all modern CPU architectures, although they refer to it in different ways and with minor differences between them, as shown for two architectures in Figure 2.3. For instance, on x86 platforms, they are called *privilege levels* [37, 38], while on ARM platforms, they are called *exception levels* [39]. In the following, we use the term *privilege level* as the generic permission state that the CPU can track. Note that this is not the same as the security context defined above, which is the smallest TCB possible for an application.

Both x86 and ARM platforms implement four privilege levels [37, 38, 39]. Figure 2.2 depicts how software is allocated to different levels in a typical deployment scenario. For the same component, software running on more privileged levels is depicted underneath software running on less privileged levels. Logically separate software modules running at the same

[2] While common, this is not the only possible model. See CHERI [36] for a notable exception.

privilege level are depicted alongside each other. Examples of this can be seen in Figure 2.2 for both the CPU and the GPU. System software, which includes the Operating System (OS) and the hypervisor, executes on the CPU on higher privileged levels than the user applications. It is used to manage peripherals, the filesystem, and perform several other critical system tasks. Isolating it from the rest of the applications thus effectively removes them from the system software's TCB. This separation makes it possible for system software to manage the application's execution and further enforce isolation between them. Having system software at a more privileged level compared to the rest of the applications has the advantage of simplifying the CPU hardware implementation, as some of the logic necessary to achieve isolation between applications running on the least privileged level can be delegated to system software rather than being the responsibility of the hardware.

The main challenge in supporting privilege levels is that, from the point of view of the CPU, software is nothing but a sequence of instructions, one executing after another. This has two implications with respect to how permissions are enforced to isolate the various contexts executing on the CPU. First, TCB separation can only be enforced at the instruction level. This means that given that the CPU can recognize that an instruction from a privilege level is executing, TCB isolation is obtained by blocking some instructions from executing if they try to violate that privilege level's permissions. Second, while at higher levels of abstraction instructions can be logically grouped into mutually distrusting security contexts, at the hardware level it is intractable to distinguish between these contexts simply by analyzing some property of the instruction stream. Next, we discuss how each of these two problems, enforcing isolation based on privilege level and tracking the execution context, are solved in modern CPU architectures.

Isolating Privilege Levels. In line with their hierarchical organization, more privileged levels can usually access (read/write) their own memory and that of lower-privileged levels. Enforcing these access privileges is achieved through a cooperation between hardware and software. Initially, the system boots into the highest privileged mode. The software in this mode (usually system software) instructs the CPU about the mapping between memory regions and privilege levels. This is done either through a mechanism called memory paging, which we will discuss in Section 2.2.2, or by specifying the expected privilege level for each memory range on dedicated CPU registers. The mapping between memory regions and privilege level can be changed at runtime, but software cannot re-assign or modify memory assigned

to higher privileged levels. A common mechanism to access peripherals is to *memory map* them, meaning that system software is able to access peripherals by referencing determinate memory regions. Memory isolation across privilege levels hence also makes it possible to isolate peripheral access depending on the privilege level. By making these memory regions accessible only to high-privileged software, lower-privileged applications are prevented from directly accessing peripherals. Low-privileged software contexts can still access peripherals, but to do so, they need to cooperate with system software, allowing it to implement fine-grained access control on them.

Recall that the CPU can only enforce the permissions at the instruction granularity. Thus, when executing an instruction that accesses memory, the CPU checks that the current privilege level is privileged enough to perform that memory access. The check is done by comparing the permission level of the to-be-accessed address, assigned with the mechanisms described above, with the privilege level of the current execution context. If they match, or the current execution context is more privileged, the instruction executes successfully. Otherwise, the instruction execution is aborted, and an exception-handling routine is executed to handle the faulty access.

Privilege Level Transitions. Hardware and software cooperate to provide a safe transition between privilege levels. A safe transition guarantees that less privileged execution contexts cannot tamper with the integrity of code in more privileged execution contexts. This entails, for example, that code is not unintentionally executed with elevated execution privileges and that control flow is not hijacked across privilege boundaries. Even slight deviations from the intended control flow are particularly dangerous in this case, as they could lead to performing actions that privileged software would not have otherwise performed.

The CPU exposes an interface to software to configure entry points into different privilege levels. Transitions across privilege levels can then only happen from those entry points. The entry points sanitize the inputs and perform several consistency checks. Recall that privilege separation mechanisms are meant to remove unprivileged software from the system software's TCB. Thus all calls from unprivileged software to privileged software are assumed to be untrusted. As a consequence, ensuring that every privilege level transition goes through a predefined entry point is vital for the correctness and security of high-privileged software. The entry points, however, are only needed when elevating privileges, while when

going from more privileged levels to less privileged ones, software is allowed to resume the execution from any point of the program.

At least two types of transitions from low-privileged software to high-privileged software are supported on different architectures: synchronous and asynchronous. Synchronous transitions are also known as *system calls* and occur when low-privileged software needs system software to perform some actions for it. Asynchronous transitions occur whenever an exception or interrupt occurs that is configured to be handled by system software. Both in the case of synchronous and asynchronous transitions, the CPU determines the appropriate entry point and then executes it, all while changing the privilege level. Note that in any case, an attempt to execute code from a higher privileged context without using the specified entry point is blocked by the CPU (by forwarding execution to an exception handler to handle the fault).

Both x86 and ARM allow to specify multiple entry points for system calls and interrupt/exception handling. On x86, the system call entry points can be specified both in the global descriptor table (GDT) and the local descriptor table (LDT) [37], while entry points for interrupts and exceptions are assigned as *vector numbers*, and their entry points are defined in the interrupt descriptor table (IDT) [37]. The GDT, LDT, and IDT are simply a range of memory that the CPU interprets in a special way. The OS configures the CPU cores instructing them on where it stores these tables in memory by saving their base pointer address in specific CPU registers: the GDTR, the LDTR, and the IDTR, respectively. Only the most privileged level is allowed to change the value of these registers. When an interrupt or exception occurs, the vector number of the interrupt or exception determines the offset through which the IDT is accessed. That offset then contains the address of the routine that will handle the interrupt. To return from an interrupt-handling routine to the previous execution context, software then executes the `iret` instruction. System calls instead happen by explicitly redirecting control flow to one of the entries of the GDT or LDT. Instructions that can have as target one of these entries are: `call`, `jmp`, `sysenter`, or `syscall`. To terminate a system call, a return to a lower privilege level can be executed with one of the following instructions: `ret`, `sysexit`, `sysret`.

On ARM architectures, context switches follow a similar procedure, but the process and the structures involved are slightly simpler. On AArch64 [39], entry points are stored as exception vectors. Each exception level has a register named `VBAR_ELx` (where $x \in [0,3]$), which stores the base pointer for the vectors of exception level x. To handle exceptions and interrupts, the CPU sets the program counter to an offset from `VBAR_ELx`.

The offset is calculated as a multiple of the vector number of the exception/interrupt to be handled. As opposed to x86, in ARM, system calls are handled in the same way as exceptions. The instructions used for system calls (`HVC`, `SMC`, and `SVC`) cause an exception with a predefined number reserved for system calls. While handling this system call "exception", the CPU changes the exception level and redirects execution to the instructions present at the offset of `VBAR_ELx` related to the system call vector number. Returns to less privileged exception levels are performed using the `ERET` instruction.

2.2.2 Virtual and Physical Addresses

Peripherals and memory ranges in the system are addressed through *physical addresses*. For instance, a range of the physical address space is generally reserved for accessing DRAM. When the CPU executes an instruction referencing a physical address in the DRAM range, it fulfills the instruction by issuing a memory request to DRAM. For a memory read, DRAM receives the physical address as part of the request and then returns the memory value that was last written at that physical address. Note that memory transactions usually happen at the granularity of a cache line (commonly 64B).

In terms of security, dividing the privileges only in hierarchical levels, as described before, would make it impossible to isolate multiple security contexts running at the same level. For instance, as shown in Figures 2.2 and 2.3, user-level applications are meant to execute with the least level of privilege. However, if they are not isolated from each other, they would be able to access and modify each other's memory. This is undesirable, both in terms of usability and in terms of security. Usability-wise, without separation, the various applications running on the system would need to coordinate to agree on memory ranges that they can each use Security-wise, this makes it possible for every application to access each other's memory, making it trivial for any malicious application in the system to leak or tamper with the data of any other application executing on the system.

Using virtual addresses solves these usability and, more importantly, security issues. Virtual addresses create a level of indirection between memory accesses and the actual accessed address. Whenever a virtual address is accessed, the CPU first translates it to a physical address and then performs the memory access to that physical address. The translation is done through a mechanism called paging, which is managed by the OS. Note how, like the privilege separation mechanism described above, this

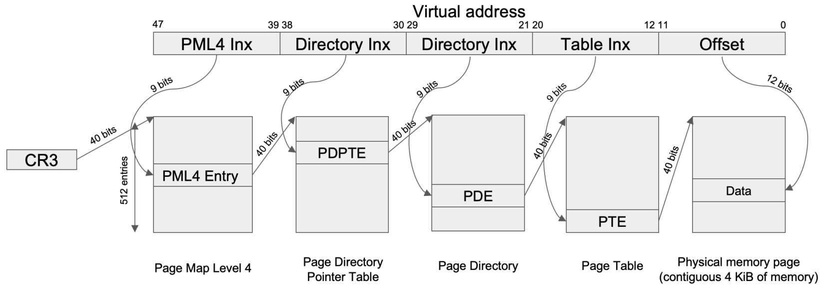


Figure 2.4: *4-level virtual address translation lookups performed on an x86 CPU to translate a virtual address into a physical one. The tables contain only physical addresses, and the virtual address is used as an index to these tables. The CPU performs this translation in hardware every time a virtual address is accessed.*

isolation mechanism is also achieved through a combination of hardware and software.

We do not delve deeper into the advantages of virtual addresses in terms of system utilization but we briefly discuss how they help in terms of isolating applications at the same privilege level. Assume that two instances of the same applications are running at the same time. These will try to access the same addresses, and if they were writing to physical memory, they would sooner or later modify the same variable and corrupt each other. With virtual memory addressing, the OS can set up translations to non-overlapping physical addresses for each application. Therefore, even if the same virtual address is referenced by both, they will access physically different locations in memory. As long as the OS maps the virtual memory to separate physical memory ranges for each application, it is impossible for one application to access the memory of another.

As system software manages the virtual memory translation, it is always capable of accessing the memory of lower-privileged execution contexts. Thus, the TCB of the OS does not include any application, but the TCB of each application contains the application itself the all of the software running on the more privileged layers above. ARM and x86 CPUs support several memory paging modes. Here, we describe the most used one in x86 CPUs to handle virtual-to-physical address translation. It uses a 4-level hierarchical lookup table structure to perform the translation. The base of the first table of this structure is stored in the CPU's `CR3` register. Because of that, system software can store multiple of these structures in memory, and

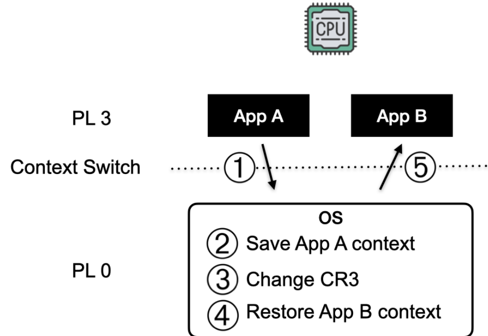


Figure 2.5: *Context switch steps performed on an x86 CPU to support multitasking.*

when a different translation is desired, say for a different application, the OS just needs to replace the base pointer in the `CR3` register, and the CPU will use the new mapping. The final address is obtained by using different segments of the virtual address as indexes to the tables, as depicted in Figure 2.4.

2.2.3 Using Privilege Levels and Virtual Addresses

To see how the two mechanisms presented above, privilege level separation and virtual addressing, work together to allow applications to execute on a separate TCB, we discuss how an OS manages multitasking and switches between applications on a modern x86 CPU.

Particularly, we do so by going through an example in which two applications, *App A* and *App B*, are executing on the system. As mentioned above, three events can cause a context switch: system calls, interrupts, and exceptions. For illustration purposes, we will describe what happens when the scheduler stops an application to execute another one. We depict the various phases that occur during a context switch in Figure 2.5 and describe each of them in the following list, where each number in the list corresponds to the same number in the figure:

① A timer interrupt previously set by the OS scheduler is received while the core is executing App A. Upon receiving the interrupt, the CPU flushes its execution pipeline and starts fetching instructions from the OS entry point of the interrupt. The entry point address is specified in the IDT, alongside the privilege level that should be set when handling the interrupt. Before executing the first instruction of the

entry point of the interrupt, the CPU switches the current privilege level to the number indicated in the IDT entry. As part of the context switch, the CPU pushes into the stack the instruction pointer of the next instruction to be executed in App A. This will be used later when the OS wants to resume the execution of App A.

② The OS saves the context of App A so that it can be used later on to be restored. This includes saving the register values that the App had at the moment of the interrupt and the value of the next instruction address that was pushed into the stack by the CPU.

③ The OS decides that it wants to now resume execution to another application that was paused before: App B. It recovers the physical address of the base pointer of the paging structure of App B and stores it in `CR3`. The CPU is now using App B's virtual address space.

④ The context of App B is now restored, register values are written back as they last were when App B was paused, and the address of the next instruction to be executed for App B is pushed into the stack.

⑤ The OS executes a return from the interrupt, using the `iret` instruction. This causes the CPU to switch back the privilege level to privilege level 3, which was the level of App B when it was interrupted. The CPU starts fetching and executing the instructions of App B.

This process ensures that multitasking happens transparently from the point of view of each application, as the context is always perfectly restored. Finally, note that the registers of the CPU which can tamper with the isolation between applications and OS (i.e., IDTR) are not accessible by the applications, and ditto for the CR3 register which could allow an application to modify its virtual addresses. These two mechanisms thus ensure that no application can access other applications' memory and tamper with the OS execution.

However, the OS not only can modify the mapping of every application but can also modify their register values and skip or modify any code or data of the application. While this trust in the OS is needed with the system design we described so far, it is not always desirable, as we will see in the next section.

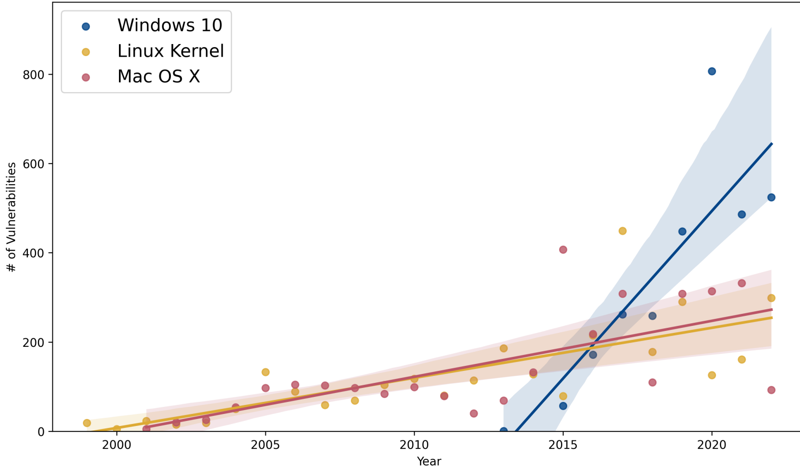


Figure 2.6: *Total reported CVEs (irrespective of severity) by year for three popular Operating Systems: Linux, Windows 10, and Mac OS X. Data from [41]. The figure also depicts a linear regression model fit over the data points for each OS to highlight the trend over the years.*

2.3 Trusted Execution Environments

System software nowadays is packed with functionality and complexity, with more being introduced with every update. Tasks such as memory management, file systems, drivers, and firewalls all execute as part of system management software, be it the OS or the hypervisor. Albeit establishing a precise model between the complexity and the number of bugs or vulnerabilities in a codebase is not trivial [40], intuitively, the higher these metrics, the higher the probability that at least a critical vulnerability will be present in the codebase. The number of Common Vulnerabilities and Exposures (CVEs) identifiers released for various OSs is a relevant proxy to the number of vulnerabilities present in their codebases. Figure 2.6 shows the reported CVEs for Microsoft's Windows 10, Apple's Mac OS X, and the Linux Kernel. Despite them being from different vendors, all these OSs exhibit an upward trend over the years in the number of reported CVEs. On the other hand, as we saw before, every application running on the system has its OS and the hypervisor in its TCB. While malicious applications are isolated by the OS, chances are that they

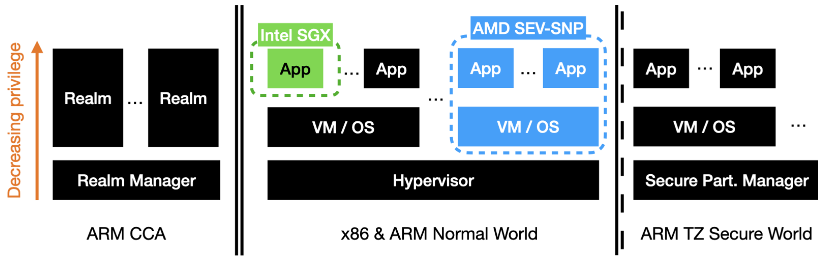


Figure 2.7: *Isolation boundaries and software allocation for TEEs across three architectures: Intel x86, AMD x86, and ARM. The figure shows both a baseline without protection and four TEEs: Intel SGX, AMD SEV-SNP, ARM TrustZone, and ARM CCA. Each block in the figure represents an isolated software module. Vertical separation is achieved through privilege level mechanisms (cf. Section 2.2.1), while horizontal separation is enforced for lower levels by software executing on the level above. Double lines (|| and |¦) only apply to ARM. Particularly, |¦ applies to ARM TrustZone, and represents one-way isolation: secure word software can access the normal word memory, but the opposite is not possible. || applies only to ARM CCA and represents full isolation: Realms cannot access the rest of the system, and the rest of the system cannot access the Realms.*

can escalate their privileges by exploiting a bug in the OS or hypervisor, then gaining complete control of the system.

Hardware supporting Trusted Execution Environments (TEEs) pushes additional primitives in hardware so that hypervisors and OSs do not necessarily need to be present in every TCBs. Different flavors of TEEs exist and have been proposed both by industry and academia [9]. Here, we discuss four major commercial ones: Intel SGX [10], AMD SEV [42], ARM TrustZone [8], and ARM CCA [12]. We present an overview of these TEEs in Figure 2.7. Particularly, the figure shows what parts of the software stack these TEEs are able to isolate. We generally refer to the isolated execution context enabled by these TEEs as *enclaves*.

Among the mentioned TEEs, Intel SGX is the one that allows to reduce and isolate the TCB the most. The enclave in Intel SGX is formed by a single application. As Figure 2.7 shows, the CPU is able to track a single application, which can then execute safely even if any other application and system management software is compromised. However, this comes at the cost of expressibility, as interaction with system software is reduced,

and applications need to be carefully refactored in order to safely utilize SGX.

AMD SEV-SNP instead isolates entire virtual machines (VMs). Thus an enclave in AMD SEV-SNP is a whole VM. This is the same level of isolation provided in software by the hypervisor, but the advantage is that by using hardware primitives, the hypervisor is removed from the VMs' TCB, as shown in the figure by the SEV-SNP boundary. In terms of programmability, this additional layer of isolation comes almost for free, as the same code that runs on a normal VM can run in a SEV-SNP VM.

While both Intel SGX and AMD SEV-SNP specialize the execution context so that the CPU can isolate lower privileged layers from higher privileged ones, ARM follows a different philosophy. Particularly, it allows to manage separate address spaces so that the entire stack can be replaced if execution on a separate TCB is desired. ARM TrustZone, divides the address space into two: Normal Wold and Secure World, with the latter being the enclave. Both these spaces execute full software stacks, including possibly a hypervisor. The idea in terms of TCB reduction is that, generally, the Secure World executes a smaller code base. These two address spaces are also hierarchical, as the software in the Secure World can access (read and write) the memory of the Normal World, but the software executing on the Normal World cannot access the Secure World memory.

ARM CCA splits the address space in an additional range, which is neither accessible by the Normal World nor the Secure World. This allows the definition of *Realms*, whose isolation is enforced by the Realm Manager (RMM). The realm manager is intended to be a thin software layer, and realms can be simple applications or full VMs. So in a sense, the CCA extension gives a similar primitive as AMD SEV-SNP, with the additional requirement of having to include the RMM in the TCB of every Realm. Realms are meant to be separate enclaves in ARM CCA, although effectively, they all share the same RMM in their TCB.

Throughout the rest of the thesis, we focus mostly on Intel SGX and AMD SEV-SNP, so we do not delve deeper into the ARM inner workings here. Next, we describe in more detail three crucial aspects of Intel SGX and AMD SEV SNP: how they keep track of a more specialized execution context, how they provide memory isolation, and how they provide attestation.

2.4 Intel SGX

Intel Software Guard eXtensions (SGX) is a novel Trusted Execution Environment technology that introduced processor extensions, which

allow for processor-supported application isolation and attestation [43]. As we mentioned in the previous section, software executed in SGX enclaves is isolated from all other software running on the system, including system management software. Enclave memory is encrypted and integrity protected whenever it is moved outside the CPU. These mechanisms are used to protect against a physical attacker that can tamper with DRAM or the memory bus. Integrity protection also includes freshness information to prevent replay attacks.

Like in classical applications, the OS remains in charge of managing the enclave's memory through memory paging. The OS is responsible for starting and scheduling enclaves but should not be able to interfere with their execution or compromise the integrity and confidentiality of their data. SGX further supports attestation, through which other enclaves or remote parties can verify enclave code and establish secure channels with enclaves. Finally, enclaves can seal data to disk using CPU-generated enclave- or developer-specific keys. SGX was therefore designed to operate under the model of a local physical adversary, who is in full control of the OS and can schedule and interrupt the execution of enclaves. The physical attacker is, however, assumed not to be able to physically compromise the CPU die.

Next, we briefly explore the lifecycle of an SGX enclave, how transitioning between enclave and non-enclave mode works, and how enclaves are attested. For a more thorough discussion of these steps, we refer the reader to Costan et. al [43].

2.4.1 Enclave Lifecycle

A key design decision in SGX relates to how much of the management of the enclave memory is offloaded to the CPU as compared to what is kept as the responsibility of the OS. As we discussed, offloading isolation enforcement to hardware increases its complexity, but the OS is untrusted in SGX's attacker model. SGX strikes a middle ground by assigning the heavy lifting to the OS while leaving the CPU "only" the responsibility to verify that the OS is not being malicious. Particularly, the OS is kept in charge of managing the virtual memory mapping for the enclaves, just as if the enclave was any other normal application. In this model, memory can be shared through the virtual memory system between an enclave and a non-enclave application, and the OS can still swap pages out to disk if it needs to free up system memory. To enforce isolation, the CPU makes an enclave's memory inaccessible from any execution context but the one of the enclave that owns the memory. For this to be possible, the CPU needs to track the virtual address mapping assigned by the OS and ensure that it

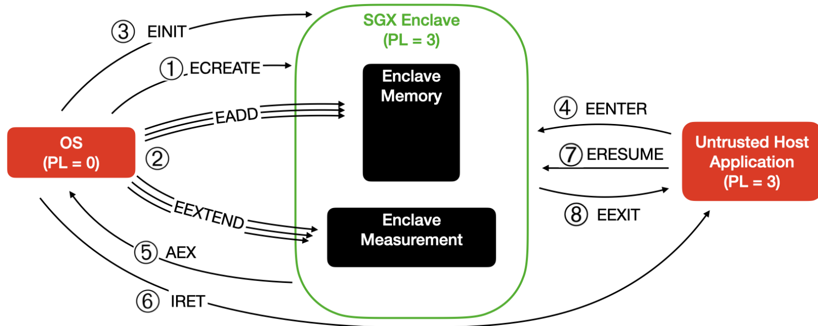


Figure 2.8: *Life cycle of an Intel SGX enclave. The OS and the untrusted host application are depicted in red, as they are not trusted in the SGX attacker model. Some instructions require privilege level (PL) 0 to be executed and can thus be done only by the OS. The enclave executes with the privileges of a user-level application (PL = 3).*

stays consistent throughout the lifetime of an enclave. For enclave memory pages that the OS wants to swap to disk, the CPU encrypts the page and then passes it back to the OS. When the OS wants to restore the page, the CPU ensures that the OS assigns it the same virtual to physical mapping. These are just some examples of the operations the CPU needs to perform to guarantee that the virtual to physical mapping is not being tampered with by the OS to break either the enclave integrity or confidentiality.

As can be inferred by this brief description of the functionalities that SGX needs to support, its operations are quite involved. In fact, rather than being implemented purely in hardware, they are implemented in a special form of microcode, referred to as XuCode [44] by Intel. All the interactions between the OS and the CPU regarding SGX happen by executing SGX instructions, which are architecturally defined in the x86 Instruction Set Architecture (ISA). These instructions (and some implicit events) are then implemented in XuCode. XuCode can be thought of as a small program written in basic microcode instructions (microops) rather than in the x86 ISA. Each XuCode instruction, much like the x86 ISA instructions, is then executed by (thousands of) microops. Note that the XuCode is signed by Intel and is akin to firmware, given the architectural level at which it operates.

Next, we explain several key steps in the lifecycle of an SGX enclave. These are depicted in Figure 2.8, which shows how some of the instructions introduced with SGX are used to allow the OS to manage an enclave

memory, all while keeping it inaccessible to the OS. The numbers in the list below correspond to the numbers in the figure and describe some of the tasks performed with each SGX instruction shown in the figure:

① The OS issues the `ECREATE` instruction to initialize a new enclave. With this instruction, the CPU receives initial parameter values to initialize the enclave. For instance, these parameters define the size of the enclave, its base address in the virtual address space, whether the enclave is in debug mode, and whether it is in 64-bit mode, to name a few. This information is stored in a structure known as the SECS, which is created by the CPU as part of `ECREATE` and is stored in the Processor Reserved Memory (PRM), a range of memory reserved at boot for SGX, which is inaccessible to any system management software (including the OS).

② The OS then starts populating the memory of the enclave. Memory is added one memory page at a time using the `EADD` instruction. Among its parameters, `EADD` needs two virtual addresses (VA). The first is the VA of the source page in non-PRM memory. The second is a destination VA which is mapped to a physical address (PA) in the Enclave Page Cache (EPC), a subset of the PRM reserved to store enclaves' pages. The `EADD` implementation (XuCode) ensures that the destination VA maps to a PA in the EPC, which is not being used by any enclave. If these and other checks succeed, the CPU copies the source page in non-PRM memory into the specified destination page in the EPC. As part of this process, the CPU also records the mapping from VA to EPC PA so that it can ensure that the OS does not, at a later stage, maliciously remap the enclave memory. Upon issuing an `EADD`, the CPU extends the enclave measurement, which, as the name suggests, is a measurement over the code and data of the enclave. The measurement is a fundamental part of attestation, as we will see later. `EADD` only extends the measurement to record that a page was added but not what the content of that page is. To include the contents of the page in the enclave's measurement, the OS executes several `EEXTEND` instructions. `EEXTEND` includes 256 bytes of memory in the enclave's measurement. Thus, it needs to be called 16 times for each added 4kB page. Both `EADD` and `EEXTEND` are repeated by the OS for each page that needs to be loaded, and by the end of this operation, the initial enclave code and data will have been copied into the PRM, and their content will be reflected in the enclave measurement.

③ When the OS is done adding memory pages to the enclave, it issues the `EINIT` instruction. This instruction freezes the memory of the enclave, meaning it will not accept any more pages via the `EADD` instruction, and the measurement is finalized. The enclave is now ready to be executed.

④ The untrusted application can call the enclave via the `EENTER` instruction. This instruction is unprivileged, as opposed to the previous ones, which can only be called by software running on privilege level 0. The enclave now starts executing one of its functions, the one specified in the parameters of `EENTER`. For the same control-flow integrity reasons that we discussed for the OS in Section 2.2.1, the enclave can only be called from well-defined entry points.

⑤ Any exception or interrupts during the execution of the enclave cause an asynchronous exit (AEX) event. AEX is implemented in XuCode, as an exit from an enclave includes several operations. As part of an AEX, the CPU saves the execution context of the enclave before exiting in a part of the PRM known as the SSA. It also cleans the register status before transferring control to the exception or interrupt handler. These handlers are implemented in untrusted code, so the CPU makes sure that the handlers cannot leak data by observing the intermediate values of the enclave's registers. Finally, the CPU transfers execution to the OS handler.

⑥ One of the differences between a normal application being interrupted and an enclave being interrupted is how the interrupted execution context is restored. In a normal application, the CPU simply saves the current instruction pointer (RIP) in the stack. The handler then terminates by issuing an `iret` instruction that implicitly pops the RIP from the stack and resumes execution, as discussed in Section 2.2.3. For SGX, the RIP is sensitive, as it reveals at what point of the execution the enclave was interrupted. As a consequence, the CPU saves it in the SSA as part of the AEX together with the other register values. A RIP value is still pushed in the stack, but it is the RIP of the Asynchronous Exit handler Pointer (AEP). This is done to maintain the same semantics of a normal interrupt/exception termination. When the OS is done handling the exception/interrupt, it thus executes an `iret` which causes the AEP code to execute.

⑦ The AEP is specified as part of the enclave creation and points to a routine of the SGX SDK (present in the untrusted application process) that eventually executes the `ERESUME` instruction. `ERESUME` takes care of restoring the enclave register from the SSA, including the instruction pointer (RIP). This allows the enclave to resume transparently. Note that steps ⑤ and ⑥ can occur several times during an enclave's function call.

⑧ The enclave terminates execution. The `EEXIT` instruction is executed by the enclave causing the context to be switched back to the untrusted app. If there are any return values from the enclave call, they are copied back to the calling untrusted application.

The steps described above give an overview of how the OS cooperates with the SGX CPU implementation to manage enclaves while still not being able to access them. At their core, these properties rely on the fact that the OS cannot access the PRM. Note, however, that the PRM does not store any memory mapping, so the memory translation routine for code execution in SGX needs to verify at every access that the memory view is still the same that was set up through the `EADD` process. Finally, loading secrets into the enclaves cannot be trivially done at initialization because the OS has access to the cleartext code and initial data. Attestation, which we discuss next, allows to securely provision secrets *after* the enclave has been initialized, and thus when the OS is not able anymore to tamper with the enclave's memory.

2.4.2 Attestation

By itself, the enclave can never ascertain whether it is running in an isolated enclave environment. From its point of view, there would be no difference between the case in which it is running on a legitimate enclave and the case in which its memory is being corrupted by the OS to make it behave as if it was.

Attestation provides a solution to this problem by proving to third parties that an enclave was set up correctly. These third parties can then interact with the enclave if they accept the attestation proof. The measurement produced as part of the enclave setup, known as `MRENCLAVE` in SGX, is used to prove that the enclave was correctly initialized by the OS. The mechanism that enables attestation is simple: the CPU computes `MRENCLAVE` as memory is being added to the enclave and then signs it upon request. Since the OS does not have access to the attestation key and cannot modify the value

of MRENCLAVE (because it is kept in PRM), by checking that MRENCLAVE matches an expected value and that the signature is correct, third parties know that the expected enclave was set up correctly.

However, while the idea behind this mechanism is simple, the protocols involved are quite complex. This stems mainly from the problem of verifying that the signing key is that of a legitimate Intel CPU. We discuss how SGX solves these problems and the limits of this solution in Chapter 7. For now, we mention that the problem is treated differently depending on whether the third party receiving the attestation proof is running on the same CPU. Particularly, local attestation allows an attestation report to be verified if the enclave and the attestation verifier are running on the same CPU, while remote attestation is used when they are not.

Local Attestation. Local attestation works by running two SGX instructions which will produce the same key material only if they are running on the same CPU: EREPORT and EGETKEY. To explain how they work, take two enclaves: enclave A, with measurement MRENCLAVE_A, and enclave B, with measurement MRENCLAVE_B.

Enclave A executes the EREPORT instruction, providing MRENCLAVE_B as one of the inputs. As the output of the instruction, the CPU produces a signed attestation report of MRENCLAVE_A. The signing key that the CPU uses can only be obtained by Enclave B if Enclave B is running on the same CPU. Note that Enclave A did not get this key, but just a report signed with it. Enclave A can then send the report to Enclave B. Enclave B can then call EGETKEY, which will generate the same key that was used to sign MRENCLAVE_A if and only if Enclave A and B are running on the same CPU. Enclave B can verify the local attestation report by signing MRENCLAVE_A with its key and checking whether the signature matches.

The CPU derives these keys from secret root key material present since manufacturing on its EFuses. Since each CPU has different root keys, local attestation reports will only be accepted if coming from the same CPU. We discuss these keys in more detail in Chapter 7 when we analyze more in detail their derivation process.

Remote Attestation. Remote attestation allows to attest to a verifier executing on a remote platform. The challenge in this type of attestation is to assert that a given attestation report was signed by a legitimate CPU. There are different flavors of remote attestation. Linkable mode and non-linkable mode [45] and Data Center Attestation Primitives (DCAP) [46]. Intel can attest the authenticity of its CPUs as it knows part of the key material in them. Therefore, in linkable and non-linkable mode,

remote attestation works by having the CPU attest to an Intel Attestation Service (IAS) first and then by having the remote verifier ask the IAS to check whether a particular attestation quote came from a legitimate Intel CPU. In DCAP mode, the IAS is removed from the attestation flow, with Intel allowing customers to set up their own attestation services. Still, to bootstrap an independent attestation service, clients need to ask Intel to confirm the authenticity of a CPU in a one-time registration process.

2.5 AMD SEV-SNP

For an overview of how AMD SEV-SNP works, we refer the reader to the offical AMD documentation [11]. In this section, instead, we focus on the similarities and differences between AMD SEV-SNP and Intel SGX. In the following, we might refer to AMD SEV-SNP simply as AMD SEV.

Like in SGX, SEV enclaves' memory is encrypted, but the integrity protection provided is weaker. SEV-SNP protects the integrity and freshness of memory only against a software adversary but not against a physical adversary [11]. This is done by preventing the hypervisor from accessing enclaves' memory and by storing authentication tags for swapped-out enclave pages. A physical adversary can still corrupt the integrity of data-in-use in DRAM.

Besides this, the biggest difference between the implementation of AMD SEV and SGX stands in the fact that AMD delegates all the isolation and enforcement of SEV to a CPU coprocessor, known as the AMD Secure Processor AMD-SP and as the Platform Security Processor (PSP). As an interesting parallel to the Intel XuCode, the PSP uses a different ISA compared to the main CPU cores. The PSP is an ARM core with *ARM TrustZone*.

In terms of TCB isolation, as mentioned, SEV isolates entire virtual machines (VMs) rather than single applications, as done in SGX. Nonetheless, the process of initializing and launching an enclave resembles the process shown in Figure 2.8. The difference is that there is no untrusted application, and instead of the OS, it is a hypervisor which is allocating pages for the enclave. The hypervisor sends commands to the PSP instead of simply executing instructions. That said, the semantics of the instructions are similar. For instance, in step ①, instead of `ECREATE`, the hypervisor sends a `SNP_ACTIVATE` command to the PSP, or in step ②, pages are added using the `SNP_LAUNCH_UPDATE` command instead of `EADD`. It goes without saying that SEV also has some nuances not present in SGX. For instance, secrets are passed from the PSP to the SEV VM by using a

dedicated page which is accessible only by the VM and not the hypervisor. This page is created during VM initialization. Finally, since SEV operates at the VM level, a notable primitive provided by SEV and not by SGX is one that allows VM migration from one AMD SEV-SNP CPU to another. As part of this primitive, the PSPs in the two CPUs communicate to exchange encryption material necessary to migrate the VM. More details on the interface provided by the PSP to manage an SEV enclave can be found in the SEV firmware ABI specification [47].

AMD SEV does not provide any specialized mechanism for local attestation. Remote attestation is achieved by asking the PSP to sign the enclave measurement that was created and maintained by the PSP as pages were added to the VM, in a similar manner as done with SGX. The difference in terms of verifying the attestation is that the remote verifier does not need to contact AMD for each attestation verification. The PSP uses a public key to sign the attestation quote. On the first verification, the remote verifier can ask AMD for a certificate chain up to that key. If that chain exists, the remote verifier knows that the key came from a legitimate AMD CPU and, thus, that the PSP is protecting the VM. However, on a TCB update, e.g., when updating the PSP firmware, the public key changes, and thus a new certificate chain needs to be obtained from AMD to validate the new public key. More details on this process are discussed in Chapter 8.

Part II

Information Leakage in TEEs

Chapter 3
Frontal Attack

3.1 Introduction

While there are many TEE proposals [43, 48, 13, 49, 50, 51, 42], they are unified in their goal: providing an integrity and confidentiality oasis in an environment ruled by malicious operating systems and hypervisors. The fundamentals for this oasis are rooted in the lowest level of the computing stack: the CPU. When application security is provided through CPU primitives, the layers above need not be trusted. Intel SGX [43] is the most widely deployed among all the TEE proposals, being available in almost every modern consumer CPU Intel manufactures. It protects applications by running them in *enclaves*. SGX authenticates and encrypts enclaves' memory accesses that cross the CPU boundary and blocks any other software in the system, including OS and hypervisor, from accessing enclaves' code and data. Nevertheless, as protected as they might be, enclaves do not execute in isolation. Enclaves share resources with other applications running in the same system, particularly memory and CPU time. By design, SGX leaves the (untrusted) OS in charge of managing these resources.

However, whenever shared resources are involved, so are side channels. Researchers were quick to point out this shortcoming of SGX [43, 15, 52, 53, 17], casting doubt into enclaves' ability to provide confidentiality, one of the core TEE goals. Intel acknowledged the problem but shifted the burden of protecting against side channels to enclave developers [54]. Curbing side channels is not trivial, and in the case of SGX, it is particularly challenging due to the role the OS plays. To manage the system resources, the OS is responsible for the enclave's scheduling, memory paging, and interrupt and exception handling, to name a few. These OS capabilities, which the attacker controls, decrease the noise of traditional side-channel attacks [15, 16] and enable new types of side channels, called controlled-channel attacks [17].

The first controlled-channel attacks allowed the adversary to observe enclave accesses at page granularity (4 KiB) without any noise by merely abusing memory paging. Revoking permissions to the enclave's pages leads to page faults, which in turn give the OS attacker a trace of every page the enclave accesses. Initial defenses that worked on the assumption that the attacker would need to trigger page faults [55] just prompted the emergence of stealthier attacks that observe page metadata set by the CPU [18, 19]. In

```
static int mpi_montmul( ... ) {
    ...
    if( mbedtls_mpi_cmp_abs( A, N ) >= 0 )
        mpi_sub_hlp( n, N->p, A->p );
    else
        /* prevent timing attacks */
        mpi_sub_hlp( n, A->p, T->p );
    return( 0 );
}
```

Listing 3.1: *Protection against timing attacks in the latest version (v2.16.6 at the time of writing) of MbedTLS. The library balances branches by having symmetric execution paths.*

response to these attacks, Intel officially recommends that SGX developers place sensitive data and code within a page [56]. Controlled channels, however, do not stop at the page boundary. OS capabilities can be used to enhance cache attacks [15, 52, 53] and extract enough information from the branch prediction unit (BPU) to give the attacker a branch granular view of the victim [16, 57, 58]. As this undermines defenses against paging-based controlled channels, further defenses leveraged the coarse timing resolution of the attacker and the inability of BPU attacks to leak the target of unconditional branches [16]. Nemesis [59] later showed that it is possible to time each instruction through interrupts, invalidating the assumptions on the best temporal resolution available to the attacker. Therefore, successive defenses [60] relied upon randomizing control flow through unconditional jumps to protect enclaves.

The current understanding of the attacker's capabilities leaves the impression that as long as branches do not have observable timing differences, do not leave a different cache trace, and BPU attacks are prevented, controlled channels can be contained. As shown in the snippet of code in Listing 3.1, even widely used crypto libraries tend to use balanced branches[1] to "prevent timing attacks."

This might seem reasonable; after all, the branches in Listing 3.1 would neither be observable with page attacks, since the same function is called on both paths, nor with Nemesis, as both paths have the same instructions. We question this last line of defense by further increasing the attacker's

[1] branches that contain the very same instructions on both execution paths

resolution and demonstrating that virtually any binary with control-flow secret dependencies leaks information in SGX.

Frontal attack. For the first time, we show that when interrupts are frequently issued, instructions' execution time is correlated to their virtual address and that the fetch and pre-decode module of the CPU frontend plays a role in this correlation. Based on this observation, we construct a new attack against Intel SGX that we call the *Frontal* attack. Our attack allows an attacker to associate a measured instruction's execution time with its offset in the *instruction fetch window* and, thus, with the instruction's virtual address. The attacker can then use these leaked execution times and addresses to infer control flow and, therefore, branch-dependent secrets.

We focus on extracting branch-dependent secrets, showing that an adversary can distinguish between two code sequences executed within SGX and hence, derive the secret branch condition that led to their execution. Unlike previous attacks [59, 61], which could only distinguish between sequences of different instructions, the Frontal attack allows the adversary to distinguish between two execution sequences even if they contain identical instructions (and even identical data). These differences are observable even when the two snippets of code reside in the same cache line and are thus not susceptible to cache side-channel attacks. We show that by using the Frontal attack, the adversary can extract the correct secret from the enclave with a probability of up to 99% on our test binaries. We discuss how two different libraries, the mbedTLS library [62] and the Intel IPP [63] Cryptography library, can be exploited using our attack – showing that, with just a few runs, the attacker can recover the condition of the executed victim branches with high confidence ($> 99.9\%$), and that with a single trace, it is possible to recover a full RSA key within seconds on 65% of the runs (out of 1000). We validated our attack on all available Intel microarchitectures since the introduction of Intel SGX (up to Comet Lake at the time of writing). We show that the attack works with high probability on all tested CPUs irrespective of their microcode version. We further discuss which system configurations are better than others for the attacker. For instance, unlike in most other microarchitectural attacks, disabling hyperthreading helps the attacker.

Defenses. Given the resolution achieved with our attack, a more realistic SGX adversary model should be one that considers the instruction pointer to be available to the attacker at any time. Confidentiality in SGX can only be guaranteed in this model if secret-dependent branching is avoided

altogether, for instance, by if-conversion [64] or by writing code following data-oblivious practices [65]. These defenses are effective against any side-channel attack – including ours. However, practically deploying them is not straightforward for two reasons. First, general compiler transformations incur high-performance overheads or require developer assistance to mark secrets [64]. Second, custom data oblivious solutions are not trivial to develop correctly and require domain-specific knowledge [65].

These practical hurdles for data-oblivious code have led to several spot defenses being continuously refined based on the adversary's capabilities. We give further evidence in this chapter that these are bound to be broken whenever previous assumptions about the attacker are challenged.

In summary, we make the following contributions:

- We investigate how frequent interrupts affect instruction execution times. In particular, we show a dependency between the observed execution times and their alignments within the CPU fetch window.

- We introduce the *Frontal* attack. It leverages the dependency between execution time and virtual address to attack Intel SGX enclaves. The Frontal attack leaks fine-grained control flow in branches containing the same instructions, even if they only span a single cacheline. It can do so with more than 99% accuracy in our synthetic binaries.

- We exploit two commonly used cryptographic libraries using the Frontal attack: the Intel IPP Cryptography library and the mbedTLS library. We further test which CPUs are vulnerable to our attack and found that all available CPUs with SGX at the time of writing (up to 10th gen) are vulnerable. Newer CPUs that include hardware mitigations against Spectre [66] seem to be more vulnerable than older CPUs. We responsibly disclosed the findings to the affected vendors (cf. Appendix A.1).

3.2 Background

SGX-Step & Nemesis. SGX-Step [67] is an open-source framework that allows single-stepping through the execution of SGX enclaves. SGX-Step uses APIC timers to interrupt the enclave after every instruction and inserts custom routines in between the interrupt handler and the enclave resumption. It does not rely on any adversarial capability not given in the standard Intel SGX attacker model, as interrupt handlers and APIC timers

are controlled by the OS, which is assumed to be under the control of the adversary.

When an enclave receives an interrupt, it performs an Asynchronous Enclave Exit (AEX) and then jumps to the interrupt-vector entry defined in the interrupt descriptor table (IDT) to handle the interrupt. After the interrupt has been handled, it jumps to the address set in the asynchronous enclave pointer (AEP). The function in the AEP eventually executes the ERESUME instruction to resume the enclave [43]. SGX-Step installs a custom interrupt handler in user space to gain control as soon as possible after the interrupt. It also replaces the AEP to execute custom instructions right before ERESUME. SGX-Step uses these modified routines to store the current cycle count just before entering the enclave and right after an AEX. To interrupt the enclave at the right time, it configures a cycle-accurate APIC timer. This timer can be configured so that the execution is interrupted after a single instruction is executed inside the enclave. These changes allow an adversary to single-step an enclave and measure the execution time of individual instructions (including a constant offset by the ERESUME and AEX).

The Nemesis [59] attack exploits the fact that the interrupt timings obtained through SGX-Step are correlated with the instruction type currently pending in the CPU. Since current processors execute some instructions faster than others, the adversary can make an educated guess about the type of instruction that was executed in a single step. Based on a trace of these timings and knowledge of the binary executing in the enclave, the attacker can detect where the instruction pointer (IP) was in the enclave when the interrupt was received. Because Nemesis can only infer the instruction type, it cannot resolve the IP whenever a balanced branch is executed in the enclave.

CPU Background: The Frontend. Although the x86 instruction set architecture (ISA) is well specified [37], the microarchitecture is typically proprietary, and its details are confidential. Generally, the processor core can be split into three main parts: the frontend, the backend, and the memory subsystem. Here, we will focus on the frontend of the processor. For further information on the other components, we refer to [68].

The frontend of a processor is responsible for fetching and decoding instructions into a format that the backend understands. Modern Intel processors need to fetch a large number of macro-ops to feed the extremely performant out-of-order backend. A modern Intel core fetches 16 bytes at once [68] from 16-bytes aligned blocks, also called the instruction *fetch*

window. In x86, there is an extra step during decoding where the fetched x86 instructions (macro-ops) get translated to a different internal instruction format called micro-operations (micro-ops).

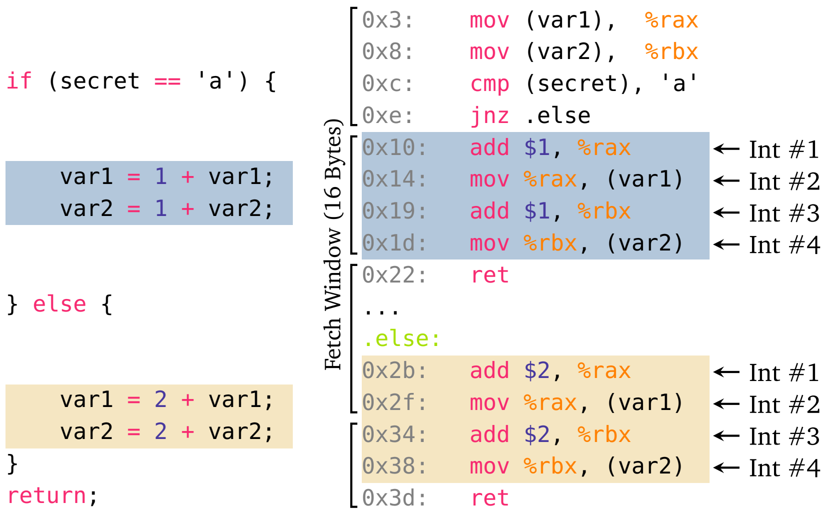
3.3 Overview of the Frontal Attack

Attacker Model. We consider an attacker that wants to leak secret data from a victim SGX enclave running on a system under their control. The victim enclave has a control-flow dependency related to the secret data the attacker wants to leak. The adversary operates under the standard SGX attacker model [43]. That is, they control the entire software stack, including the operating system (OS), on the machine in which the enclave executes. Since the attacker controls the OS, we assume they can disable any CPU core to reduce noise or prevent the scheduler from running tasks on a particular core. However, the CPU package is not physically compromised. We assume that the secret that the enclave holds was remotely loaded after a successful attestation. Otherwise, if the secret were to be contained in the enclave code, it would be trivially available to the OS.

Attack Overview. We introduce our attack through an example code snippet that we show in Figure 3.1a (C code) and Figure 3.1b (x86 assembly). The code fits in a single cacheline and has a branch whose target depends on a secret value. On both branches, the code contains the very same instructions and writes to the same memory addresses. Thus, we expect its execution time to be independent of which branch is taken and hence not to have any correlation with the secret input.

However, when the above sequence is run within an SGX enclave, our attack shows that a local attacker can learn which branch was taken and, therefore, derive the secret value of the branch condition. Our attack leverages two main observations. First, even if the branches have the same instructions, they are often aligned differently within the fetch windows (Figure 3.1b) – in our experiments, this alone did not produce observable differences in the execution times (cf. Section 3.4). Second, if the execution of both branches is frequently interrupted, the difference in their alignments w.r.t. the fetch windows will cause the CPU to fetch instructions at different times (Table 3.1), resulting in a measurable difference in the execution times of the instructions and therefore of the branches (cf. Section 3.4).

To give an insight into why interrupts lead to a successful attack, we show which instructions are fetched by the CPU when the execution is interrupted after each instruction. There are two main factors to consider:



(a) *C source code of a secret-dependent branch*

(b) *Compiled version of the secret-dependent branch on the left*

Figure 3.1: *A secret-dependent branch in C and x86 assembly. Both branches in the assembly code fit within the same cacheline (64B). The virtual address of the instructions is reported on the left. Note that while the branches are instruction-wise identical, their instructions get grouped differently by the fetch window (which always starts at multiples of 16B).*

which instructions among those already in the pipeline are retired when an interrupt is received and how execution is resumed after an interrupt. Intel guarantees that only the oldest pending instruction in the reorder buffer is retired [2] before the interrupt is handled [67]. In out-of-order processors, other instructions might have already been executed, but none of these will be retired. To resume execution after the interrupt is handled, the CPU needs to fetch the instruction sequence starting at the current program counter. However, while the program counter can, in general, have any value, fetch windows are statically aligned at 16 bytes code blocks [68]. Assume that the program counter falls 5 bytes after the start of the fetch window. Those initial 5 bytes will be fetched only to be then discarded by the frontend. Thus out of 16 bytes fetched, only 11 are usable. Now assume

[2] Or discarded, if it raises an exception

that the same instruction sequence begins 10 bytes after the start of the fetch window. Instead of 11 bytes as before, there are only 6 bytes that can be decoded, meaning we now need two fetch windows (and hence two cycles) to decode the same number of instructions as we did before in just one fetch window. Alignment w.r.t. fetch windows can, therefore, change the order in which instructions are forwarded to other stages of the CPU and ultimately populate the pipeline. To help clarify this point, for both branches of our example code, we show in Table 3.1 which instructions are fetched after every interrupt.

In principle, given the same system conditions, a particular instruction should exhibit the same time distribution at different virtual addresses. However, we experimentally observe that depending on the alignment within a fetch window and the number and type of instructions present around them, some instructions consistently take longer to execute than others. In Section 3.4, we provide more details on which alignments of instructions produce measurable execution time differences. This observation hence allows us to associate the measured instruction execution time with the alignment in the fetch window and, therefore, with the instruction virtual address (i.e., with the instruction pointer). These leaked execution times and addresses can then be used to infer executed branches (e.g., when they depend on the secret value). In this work, we focus on the use of our attack in the context of secret-dependent branching. In particular, for the scenario given above in Table 3.1, when enough `mov` are fetched after a `mov` in the branch, the interrupt latency is measurably different. In our example, we measured interrupt #2 in the table to be faster if the code is executing in the "else" branch, as compared to the "if" branch, despite the fact that we are interrupting the *same instruction under the exact same system conditions*.

Let us again consider Figure 3.1b. By running SGX-Step, we can time all instructions by stepping through them one by one. As a consequence of the observations made above, we will observe two scenarios for the 6th instruction measured, which is the instruction at address 0x14 or 0x2f, depending on the secret value. If the interrupt is "slower" (compared to the others measured), we must be executing the `mov` at address 0x14. Inversely, if the interrupt is "faster", we must be executing the `mov` at address 0x2f. Since the control flow of the program depends on the secret, this allows us to recover its value and hence break the SGX confidentiality guarantees.

The snippet presented in Figure 3.1 produces distinguishable timings for the first `mov` instruction inside the branch. We were able to use the timing difference to predict the secret with $\geq$ 65% accuracy. By adding

Table 3.1: *View of how instructions are batched into fetch windows when the enclave resumes execution, according to which branch is executing. If an instruction crosses a fetch window boundary, we assume it is decoded together with the instructions in the following window. The interrupts refer to the instructions in Figure 3.1b.*

	If			Else			
Interrupt #1	add	mov	add	add			
Interrupt #2	mov	add		mov	add	mov	ret
Interrupt #3	add			add	mov	ret	
Interrupt #4	mov	ret		mov	ret		

three more `mov`s after the branches (which are executed by both paths), we were able to obtain success rates > 90%. The attack presented above illustrates how fully balanced branches actually produce secret-dependent timings when interrupted frequently. Given that this side channel is due to the design and behavior of the CPU frontend, we name our attack the *Frontal* attack. In the following sections, we will analyze our attack in more detail.

3.4 Frontal Attack Profiling

In this section, we provide more detail and clarification to that help in understanding under which circumstances the Frontal attack works. More specifically, we ask and answer the following questions: (i) are the interrupts required for the attack to be successful? (ii) what are the effects of the fetch window alignment/instruction address on the attack? and (iii) which instructions produce observable timing differences?

To answer these questions, we perform experiments over the code snippet shown in Listing 3.2. Similar to the code in Figure 3.1, this code snippet contains two perfectly symmetric branches depending on a secret. It still consists of two perfectly balanced branches but differs in that, now, each branch contains 25 sequences of `add-mov` instructions. We chose this longer code sequence since it produces timing differences that are more clearly above the noise floor than the code in Figure 3.1 and, therefore, better illustrates timing and alignment effects under different experiment configurations. Namely, code sequences that include several `mov` instructions, like the one in Listing 3.2, are particularly susceptible to the Frontal attack and allow us to extract the secret branch condition with an accuracy of at least 99%, whereas with shorter sequences that contain few

```
              .align (x - 0x4)
x - 0x004:      cmp (secret), 1
x - 0x002:      jnz .else
              .if:
              .rept 25
x + 0x000:      add %rax, %rbx
x + 0x003:      mov %rcx, (var1)
              .endr
x + 0x190:      ret
...
              .align y
              .else:
              .rept 25
y + 0x000:      add %rax, %rbx
y + 0x003:      mov %rcx, (var1)
              .endr
y + 0x190:      ret
```

Listing 3.2: *ASM Code with high attack success probability, which we use to profile the attack. The `.rept 25 ... .endr` assembler directive repeats the instructions within the block 25 times, leading to an address of `x+0x190` for the `ret` instruction.*

`movs` (like the one in Figure 3.1), this accuracy drops to $\geq 65\%$. We discuss this effect in more detail later in this section.

3.4.1 The Role of Interrupts

To analyze the effect of frequent interrupts on the behavior of the processor, we measure the execution time of our test code snippet (Listing 3.2) with and without interrupts.

Outside SGX without Interrupts. We first measured the overall execution time of the code snippet outside SGX without interrupts. We executed the code with two billion independent random inputs, and we observed no significant correlation between execution times and the branch that was executed (Pearson's coefficient $= -2.51 \cdot 10^{-5}$). An approximate distribution of this measurement is shown in Figure 3.2.

In SGX without Interrupts. In order to exclude any effect due to SGX, we further measure the overall execution time of the code within an SGX enclave, again without interrupts. Note that SGX does not provide any way

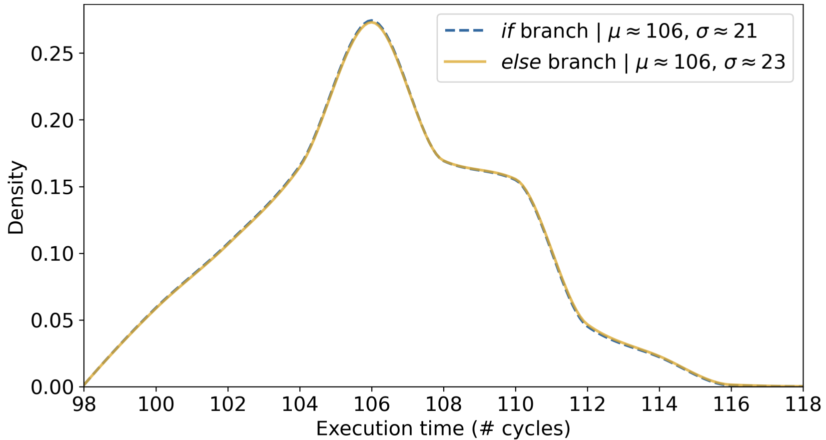


Figure 3.2: *Distribution of the overall execution time of the branches in Listing 3.2 when run outside of SGX without interrupts (computed from* $2 * 10^9$ *samples).*

to get a precise timer (cf. Chapter 2), so we have to measure the execution time from the untrusted app.

We perform this measurement using three different methods. All methods use the same code snippet in a loop, but they differ in how the measurement is collected and where the loop is executed. We do this to filter out any effects of the enclave entry and exit operations. First, we measure a whole enclave call from the untrusted app. Multiple measurements are collected by having a loop in the untrusted app. Second, we run the loop entirely inside the enclave and collect the iteration execution time with two `ocalls` to the untrusted app. The two `ocalls` are done at the beginning and the end of each loop iteration. Third, we use a similar setup as the second method, but instead of performing `ocalls`, the enclave samples the value of a counter stored in shared memory. A thread of the untrusted app increments the counter in a loop, thus simulating the time stamp counter, albeit at a lower precision. All three methods use an independent uniform random value as the "secret" given to the code at each iteration.

For all three methods, similar to the experiment outside of SGX, we observed no significant correlation (Pearson's coefficient $\approx 10^{-2}$ with 10^6 runs) between the execution time and the secret provided to the enclave.

In SGX with Interrupts. We now investigate which effects frequent interrupts have on the execution time of the code. We execute the same code snippet as before, but we interrupt it after each instruction. Upon each interrupt, the CPU performs an asynchronous enclave exit (AEX), handles the interrupt, and then performs an `ERESUME` to resume the enclave execution. Such an experiment would normally require very fast and extremely precise interrupts, which is usually hard to achieve. However, in the case of a victim code running within SGX, we can use SGX-Step [67] to single-step through each instruction and collect its execution time. Given these interrupts, we can not only measure the overall execution time but also the execution time of each instruction. This means that in each run of our code, we obtain 51 measurements [3].

We then analyzed whether any of the 51 measured instruction execution times correlate with the executed branch. We observed a *strong correlation* between the timings of most of the instructions and the branch they belong to. The first 10 `mov` instructions in the branch turned out to be a more robust indicator of which branch was taken, but all the other instructions belonging to the branch showed some correlation, albeit a weaker one [4].

As in Section 3.3, we observed the execution time of the first `mov` in each branch to be faster or slower, depending on the branch it belongs to, with a difference between the slower and faster `mov` of around 100 cycles. This observation allowed us to set a timing threshold with which we could, with up to 99.9% accuracy, determine which branch was taken and therefore determine the secret branch condition.

We stress again that the two branches are instruction-wise identical: the instructions they contain *and their inputs* are the same. This is especially important because it highlights the fact that the timing difference is due to the way the instructions are executed and not some external system state. For instance, the difference *cannot* be due to the state of the cache, the state of the branch predictor, or in general, to some speculation decisions made by the CPU. If the cause of the differences were to be due to any of these factors, we would expect two key differences. First, as we choose secrets at random, these effects would manifest with equal probability in any of the two branches. Second, we would expect the experiments in which we do not interrupt the code to also show some bias. However, we see a clear

[3] There are 52 instructions in Listing 3.2; however, the first `cmp` and `jnz` get macro-fused into one instruction which *usually* is not split again by interrupts (cf. Section 4.6.1).

[4] The timings of the initial `cmp` and `jnz` were independent of the executed branch – only instructions within the branches were correlated with the secret.

bias in one of the two branches, and the interrupt-free runs showed no correlation with the secret.

> **Observation 1.1:** When code execution is frequently interrupted, the execution times of selected instructions depend on their location in the victim binary and, therefore, on their virtual memory address.

3.4.2 Relationship to Virtual Addresses

While the instructions in both branches are identical, there is one key difference between them: their virtual address. Therefore, we analyze what virtual addresses make the two branches distinguishable when frequently interrupted – and to what degree. In particular, as discussed in Section 3.3, we also study how the relationship between the alignment of the branches with respect to the fetch window affects the success of the attack. As can be seen in Listing 3.2, we use the `align` compiler directive to explicitly align each branch to a given address. With `.align X`, we indicate that the code following the directive starts at the next virtual address whose lower bits are equal to `X` [5]. For example, if $X = 3$ and $Y = 2$, then the `if` branch will start at address `0x13` and end at address `0x1a3`, while the `else` branch will start at address `0x1b2`.

To evaluate different alignments, we ran an experiment to test if different values of `X` and `Y` in Listing 3.2 have any effect on the observed timing differences. We repeat the interrupt experiment described at the end of Section 3.4.1. That is, we send an interrupt to each instruction and then use the interrupt timing of one of the instructions in the branch as a discriminator to determine which branch was taken and, thus, what the secret was. We then calculate the attack success as the percentage of correctly identified secret bits. Therefore, the attack success rate will tell us how good a certain combination of the alignments `X` and `Y` are for the attack. The higher the percentage, the better an alignment combination is for the attack, while a result close to 50% indicates that predicting which branch was taken is as good as a random guess. We collect these percentages for each combination of $\{X, Y\} \in [0, 31]^2$ by running the code in Listing 3.2 1000 times with uniformly random secrets. We use the timings of the 10th instruction (5th `mov`) to discriminate between the branches. Figure 3.3 presents the result of our experiment. These results

[5] This is equivalent to combining the two gcc asm directives `.align` $(X//2^n)$ and `.space` $(X\%2^n)$ (for the biggest n such that $2^n < X$)

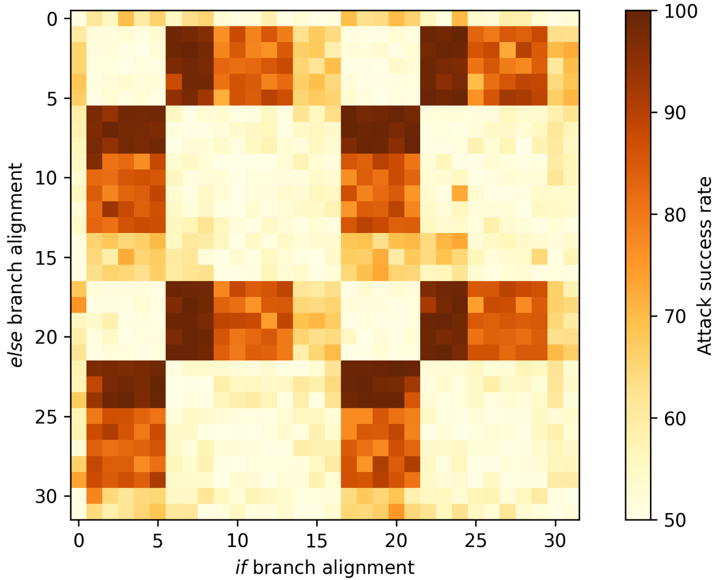


Figure 3.3: *Attack success rate depending on the alignment of the branches. The attack success rate is the percentage of correctly guessed branches by the attacker out of 1000 executed branches. The 10th instruction (5th `mov`) from Listing 3.2 is used to distinguish between both branches. The color gradient goes from darker to brighter, where darker boxes indicate higher attack success rates (up to 100%) and brighter ones lower success rates (down to 50%).*

show a clear dependency between virtual addresses and the instruction execution times.

Modulo 16. There are four main quadrants of length 16 that are essentially identical. This hints at the fact that the behavior with respect to the alignment of the two branches repeats every 16 bytes. We verified this assumption by repeating the experiment for every value of `X` and `Y` for which the two branches are still contained in the same 4 kB virtual page [6]. We observed the same pattern for all the quadrants of length 16 in this test.

[6] We did not cross the virtual page boundary because this would most likely require fetching pages that are not cached, thus introducing noise that masks the effects that we are interested in measuring.

As a consequence of this observation, when we use the term alignment, we refer to alignment modulo 16.

> **Observation 2.1:** The attack success rate depends on the alignment *modulo 16* of the two branches.

Diagonals. The attack success rate on the diagonals in each quadrant is around 50%. In the diagonals, both branches are aligned to the same value `X = Y mod 16`.

> **Observation 2.2:** Branches and instructions with the same alignment will show the same execution times.

Symmetry. The attack success rates are symmetric with respect to their diagonal, meaning that the success of the attack when the “if” branch is aligned at address `X` and the “else” branch at address `Y` is the same when the alignment of the branches is switched.

> **Observation 2.3:** Alignments X, Y and Y, X produce the same attack success rate.

Shape. Finally, we focus our attention on the alignments in the heatmap in which the success rate is above 70%. These success rates are grouped into rectangles. Within each of these rectangles, there are three regions of decreasing intensity. The most interesting alignments are the ones that give the higher attack success rates, as they allow to optimize the accuracy of the attack. The best results are concentrated on rectangles of size 3 × 5. This corresponds with the length in bytes of the two instructions within the branch in Listing 3.2. The `add` instruction has a length of $3B$, while the `mov` we use in Listing 3.2 has a length of $5B$. Unfortunately, this rule does not trivially generalize with more complex instruction size combinations.

Note that there are only a few structures in the CPU that are sensitive to the alignment of the instruction, and in particular, to their alignment modulo 16. On Skylake and Coffee Lake architectures, one of them is the *instruction pre-decode and fetch* module in the frontend of the CPU, which uses a fetch window of 16 bytes to fetch instructions from the L1 instruction cache. We cannot be entirely sure about the internal behavior of the CPU and what leads to the timing differences in the two branches. However, as discussed in Section 3.3, the different alignment changes the

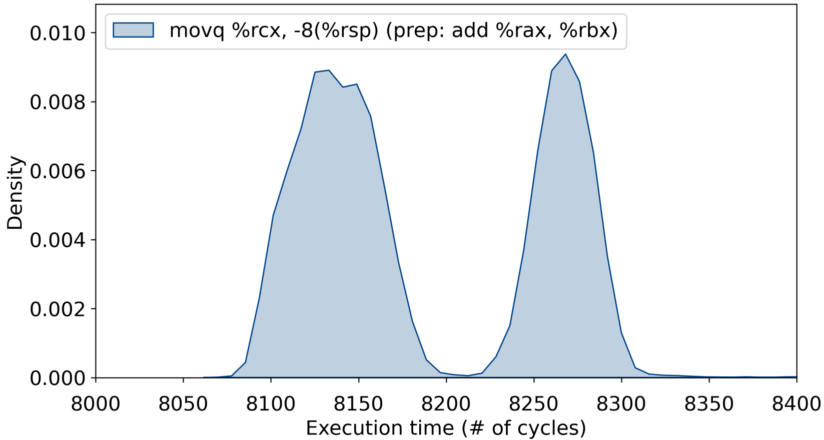


Figure 3.4: *Timing distribution of a* `mov` *to the stack when executing it in a trace containing* 100, 000 *repeated* `add-mov` *instructions (unrolled).*

way instructions are batched by the frontend and, ultimately, the timing at which they are delivered to the subsequent stages of the CPUs. The experiments presented in this section strongly suggest that these fetching differences have repercussions for the instruction's execution time. We will discuss potential causes that could lead to the observed variable timings in Section 3.7.

3.4.3 The Effects of Instruction Alignment

To study the effects of the instruction alignment, we analyze the timing distributions of a linear code sequence of 100,000 repeating `add-mov`. Note that essentially this is an unrolled loop, which, compared to a normal loop, removes the noise that the loop-control instructions would introduce. We do not envision any real code to have such a sequence of instructions, but by exploring the patterns that emerge from these instructions, we can gather several insights about how the differences in branch alignments manifest.

The timings are collected using a slightly modified version of SGX-Step, whose changes are described in Appendix A.3. The timing of each instruction includes the time to perform `ERESUME`, the time to execute the instruction, and the time required to perform `AEX`. `ERESUME` and `AEX` prepare the CPU for the enclave execution and clean the state when returning to the untrusted app. These operations take thousands of CPU cycles to complete, and this is why, despite the fact that we are measuring a single instruction,

the latencies reported in the graphs are in the order of thousands of cycles. We use two figures to illustrate different aspects of the timing latency of the same run: (i) Figure 3.4 depicts the overall latency distribution of all the `movs`, and (ii) Figure 3.5 the distribution separated by particular virtual addresses.

Distribution of Instruction Execution Times. In Figure 3.4, we present the distribution of the instruction execution times, estimated from all the 100,000 executed `mov`. The most evident feature of this distribution is that it consists of a *bimodal* Gaussian distribution. The `movs` are therefore exhibiting two different distribution modes whose peaks are, on average, around 100 cycles apart. We refer to the mode with the lower average and the one with the higher average as the *fast mode* and *slow mode*, respectively.

> **Observation 3.1:** The timing distribution of the `movs` follows a bimodal distribution. The peaks of the two distribution modes are around 100 cycles apart.

In general, we observed similar results with other instructions that access memory, such as `add` to memory. We remark here that these differences are not due to the state of the L1 data cache. We ensure this by running the victim enclave on a dedicated physical core in the system and by always performing the same operations while handling interrupts. We further verified with the `OFFCORE_REQUESTS_ALL_REQUESTS` performance counter that no extra off-core memory transactions were being performed.

> **Observation 3.2:** Observation 3.1 applies not only to `movs` but to all *memory writes*.

Instruction Execution Times by Alignment.

Regarding alignment, there is an important characteristic of the chosen instruction sequence that has not been considered in our analysis thus far. Each couple of `add-mov` in the sequence has a length of 8B, which is a multiple of 16. This implies that the `movs` *can only be aligned modulo 16 in two different ways*. In general, by testing the sequence with different initial offsets, we observed `movs` at addresses between `1 and 8` to be predominately slow and `movs` at addresses `9 to 16` to be predominately fast. We highlight that the two alignments are only *predominately* fast (or slow), and, usually, they exhibit timings from both distribution modes. We can think of each instruction at a given alignment to have a certain

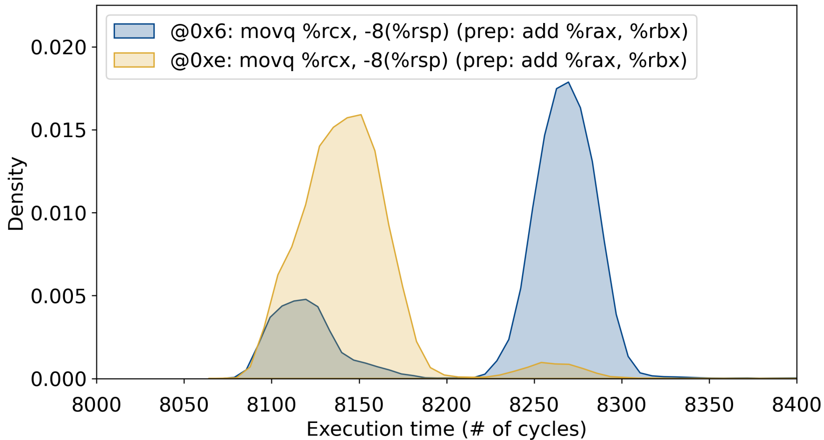


Figure 3.5: *Timing distribution of the* `movs` *from Figure 3.4 grouped by their virtual address alignment.*

intrinsic probability p to exhibit the fast mode and probability $1-p$ to exhibit the slow mode every time it executes. Different alignments have a different value of p. Figure 3.5 shows this phenomenon for two particular alignments (`0x6` and `0xe`). As can be seen, alignment `0x6` is predominately slow, but some of its timings exhibit the fast mode as well. The plots for other alignments are similar, with the only difference being the size of the smaller peaks.

> **Observation 3.3:** The alignment of the memory writes determines how their latency will distribute between the fast and slow distribution modes.

The value of p relates to the attack success rate. Say that one branch is aligned such that the measured `mov` has $p \geq 0.9$, and the other is aligned to have a $p \leq 0.1$, then the branches are easily distinguishable, and a high success rate will be observed. If one of them has $0.3 \geq p \leq 0.7$, and the other has a very small or very high p, as is the case for the distributions in Figure 3.5, then one bit can be distinguished with high accuracy, but the other will contain some errors. If the branches have a $p \approx 0.3$ and, say, $p \approx 0.7$, then both branches will be, on average, guessed better than random but will also contain errors. And finally, if both branches have a similar p, the success rate of the attacker will be negligible.

3.4.4 Requirements and Limitations

In our experiments, we only observed timing differences in branches that contain *memory writes*. Thus, at least a memory write must be present for the side channel to emerge. All the other conditions being equal, other memory write instructions we tested (variations of `mov` to different addresses and arithmetic instructions that write back to memory), excluding the `push` instruction, exhibited the very same behaviors as described so far. Notably, instructions that are surrounded by other memory writes also show a timing difference, albeit usually smaller. Furthermore, the timing distribution of a memory write is not only determined by its alignment in isolation but it is also influenced by the number and alignment of surrounding memory instructions. For instance, the more memory writes in the branch (or even right after it), the more distinguishable the distributions will be, increasing the probability of success of the attack. Another element we were able to characterize relates to the vicinity of the memory instructions with each other. Particularly, when writes are executed in a loop, the attack success probability is higher if the loop executes only a few instructions (around 10) in between writes, and the fewer, the better for the attack.

It is worth noting that simultaneous multi-threading (SMT) was a big source of noise in our experiments. When the core co-located with the victim is executing a CPU-heavy workload, we were unable to observe any significant timing difference. In general, the Frontal attack is more reliable if SMT is disabled or the virtual core co-located with the victim is idle. We speculate that this is most likely due to how the frontend handles and fetches instructions coming from different virtual cores, but possibly also to the resulting lower interference in the memory subsystem.

3.5 Frontal Attack Exploitation

The Frontal attack exploits control-flow secret dependencies. Therefore, the first step of the attack is to identify target code paths in the victim binary which execute secret-dependent branches. Several techniques have been proposed to automate finding such code paths [69, 70]. Among these code paths, as discussed before, the attacker should choose those that contain at least one memory write. Until now, we mainly focused on balanced branches, but unbalanced branches are also distinguishable with our attack. As unbalanced branches can be exploited with other attacks as well, we focus on more challenging balanced branches in our example exploits below. Balanced branches are not rare in compiled code. In fact, we found two

code patterns that commonly lead to this type of branch: slightly different return statements, and inlined function calls with different parameters.

In the following, we give examples of vulnerable branches satisfying the conditions above in two libraries: the Intel IPP Cryptography library [63], and the mbedTLS library [62]. We note that since a secret-dependent code path must be present, branch-prediction attacks can also exploit the binaries vulnerable to the Frontal attack. For instance, the examples we present below, when compiled with `gcc`, are also vulnerable to branch-shadowing attacks [16]. However, when compiling the mbedTLS library with the compiler from Hosseinzadeh et al. [60] (which is designed to prevent branch-shadowing attacks), all the branches are translated to indirect unconditional jumps, which are hitherto not vulnerable to any known BPU attack. On the other hand, we verified that even when using the compiler from Hosseinzadeh et al. [60], the branch targets are unchanged and have, in general, different alignments, thus remaining vulnerable to the Frontal attack. The attacks described in this section were performed on an Intel i9-9900KS CPU with the latest microcode available at the time of writing (`0xca`).

3.5.1 Intel IPP Cryptography Library

The Intel IPP Cryptography library is a cryptographic library optimized for Intel CPUs and advertised as constant-time [63]. However, through manual inspection, we identified several secret dependent branches in its most recent version (2.9 at the time of writing). Among these, the `l9_ippsCmp_BN` function compares two big numbers represented as arrays of integers by iterating through each element of the array. The function then terminates when a different array entry is found. It can take three different exit paths, depending on whether the first input is smaller, bigger, or equal to the second. The *smaller-than* and *bigger-than* paths are instruction-wise identical, while the *equal* path contains the same instructions as the others but in a different order. Given that the different order of instructions of the equal vs. unequal paths can be inferred with other attacks, we focus on distinguishing the *smaller-than* vs. *bigger-than* paths with the Frontal attack. With branch-prediction mitigations in place, other known attacks do not allow to leak this information, as all the paths fit in a single cache line. The exit paths contain a `mov` to memory, which we target in our attack. We did not observe any timing difference on this instruction alone, despite the fact that the paths start at different alignments; this is expected as the memory write is executed only once. However, by inlining the function in an enclave that performs a loop of at least 9 memory writes after the IPP

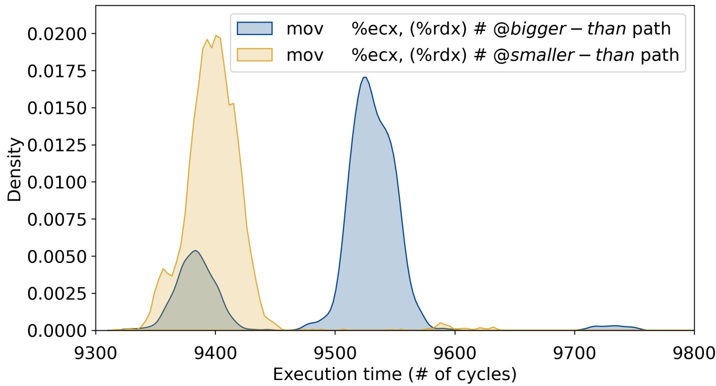


Figure 3.6: *Timing distributions of two different* `movs` *in the IPP Cryptography library's* `l9_ippsCmp_BN` *function (each estimated from* 3000 *samples). The function executes a secret dependent comparison, which can result in two balanced paths being taken: the bigger-than or smaller-than path. Each path contains a differently-aligned* `mov` *in it, whose distribution is shown in the figure.*

function call, we obtained the distributions shown in Figure 3.6. The figure shows two distributions that differ in their modality. The timing distribution of the `mov` in the *smaller-than* path has a single peak around 9400 cycles. On the other hand, the `mov` in the *bigger-than* path exhibits two modes, a small one around 9380 cycles and a predominant one at 9525 cycles, and is thus usually slower to execute than the `mov` in the *smaller-than* path. Consequently, if a measured `mov` timing is "slow," it must mean that the *bigger-than* path was executed (3% false positive). Overall, by using this comparison repeatedly with a secret bitstring as input, we were able to accurately recover 25% of the secret's bits (with 1000 function calls).

In their response (cf. Appendix A.1), Intel specified that the `l9_ippsCmp_BN` function is not used for secret-dependent computation in their architectural enclaves. However, the IPP crypto library documentation [63] mentions that its functions are commonly used for cryptographic computation and has no indication that this, or any other of its functions, is unsafe and should not be given confidential data as input.

3.5.2 Montgomery Modular Multiplication

The Montgomery modular multiplication (MM) is a fast MM algorithm often used in cryptographic libraries due to its efficiency and minimal

secret dependence. There is only a single secret-dependent branch in the algorithm: a conditional subtraction that is done at the end of the multiplication. MM is used to perform modular exponentiation, and knowing whether the subtraction was done or not leaks some bits of a secret key used in the exponentiation [71]. Some implementations, including mbedTLS as of version 2.16.6, just balance the branches by adding an else branch with a dummy subtraction in it (cf. Listing 3.1). However, this naive mitigation is still vulnerable to side-channel attacks that target control-flow secret dependencies, such as the Frontal attack. We compiled the mbedTLS library with the `gcc -O3` flag and used it inside an enclave that performs a modular exponentiation (as the MM function is not directly exposed in the library's API). The `O3` flag inlines functions when possible, so instead of performing two function calls, as shown in Listing 3.1, the binary contains two identical copies of the `mpi_sub_hlp` function. The branch condition determines which of these two gets executed. The `mpi_sub_hlp` function contains a loop with two memory writes. The loop repeats a number of times proportional to the size of the modulus of the multiplication. In Listing A.1 in the Appendix, we give the assembly code generated by the compiler for the loop we exploit. Since the two loops were aligned differently, they exhibited different timing distributions, as shown in Figure 3.7. While the differences were not as significant as seen in our controlled tests (most likely due to the fact that several instructions are executed in between consecutive memory writes), they were enough to differentiate the branches. Using Welch's t-test, we correctly classified 83% (511 out of 616) subtraction calls (whether they were dummies or not) with 99.9% confidence with just 16 repetitions of an exponentiation with the same inputs.

3.5.3 Leaking RSA Keys

We demonstrate a full end-to-end attack leveraging the Frontal attack by exploiting the function that generates a new random RSA key pair (`mbedtls_rsa_gen_key`) in mbedTLS v2.16.6. This function has several secret-dependent branches. The one we target is executed during the computation of $gcd(e,(p-1)(q-1))$, where e is the RSA public exponent and p and q are two RSA primes. Leaking $(p-1)(q-1)$ allows us to easily compute the RSA private key (as together with $n = pq$, we can solve for p and q and then compute $d = e^{-1} \mod \lambda(n)$). Control-flow leakage from the `gcd` implementation has been thoroughly studied [72, 73, 74], but it only leads to partial information recovery without fine-grained execution traces [72]. The binary `gcd` implemented in mbedTLS has a main loop that

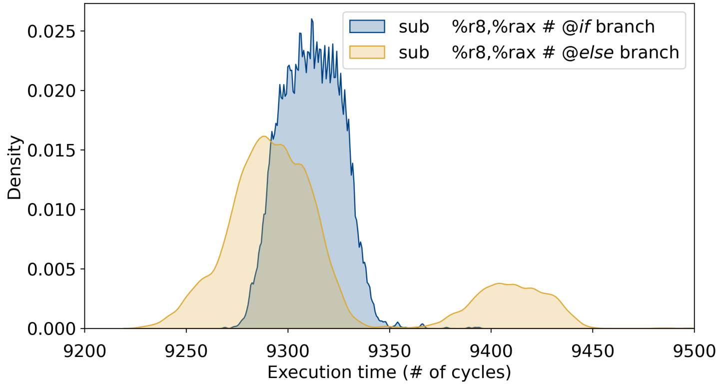


Figure 3.7: *Comparison of the real subtraction (if branch) and dummy subtraction (else branch) branches in the mbedTLS MM implementation. The two branches are identical, and both include a `for loop` that executes two memory writes (cf. Listing A.1). The graph shows the distribution of the 11th instruction in the `for loop` (a `reg` to `reg` subtraction), highlighting that as long as memory writes are present, surrounding loop instructions produce different distributions based on their alignment as well. The distributions were estimated from* 1000 *function calls, each of which has* 6 *loop iterations, resulting in* 6000 *measurements per instruction.*

removes the trailing zero bits to its operands and then has a balanced branch in which a subtraction and a `shift-right` is performed. To recover the RSA private key, it is sufficient to leak two pieces of information: the output of the function that counts the number of trailing zero-bits and the path taken in the balanced branch. We leak the trailing zero-bits by counting the number of instructions executed in the respective function, as demonstrated in [61, 74]. The result of the balanced branch is leaked with the Frontal attack. Similar to the MM attack described above, the branches need to contain inlined function calls for the attack to work. To achieve this, we modified the signature of the `int mbedtls_mpi_shift_r(...)` function in `bignum.c` to `inline int mbedtls_mpi_shift_r(...)`. Note that different compiler versions might lead to the compiler inlining this function on its own, thus producing a vulnerable binary. With this function inlined, the branches both contain the loop shown in Listing A.2 in the Appendix, leading to a differently aligned memory write depending on the branch taken. This loop within the branches is usually executed 32 times,

giving us a fairly high number of memory writes to profile. We collect and use the information from the distribution of each instruction in the loop in order to recognize which branch is being executed. The overall timing distributions are omitted here due to lack of space, but in short, some instructions look like Figure 3.7, while others look more like Figure 3.6. This means that we can classify the branch whenever any instruction's timing is in the "slow" mode of Figure 3.6 or whenever an instruction's timing is in the tail of the distributions of Figure 3.7. We executed 1000 runs, fixing the exponent to $e = 65537$ and generating a new pseudo-random key in each run. Note that since a new key is generated on each run, we cannot correlate the executions of multiple runs. In each execution, the attacked branch was executed 1018 times on average ($std = 25.40$), and on average we could not classify 89 ($std = 92.35$, median = 55) branches. This means that, on average, we would need to brute force 89 bits to recover the secret key. In practice, we noticed that since the exponent is orders of magnitude smaller than $(p-1)(q-1)$, early iterations of the secret branch are very likely not taken. Leveraging this information, we perform several guesses of the key starting from the last unclassified iteration. We assign this iteration as 'taken' and check if this results in a correct key. If not, we assign the next iteration as taken as well and repeat. This greedy approach worked on 65% of the runs and allowed us to recover the key of those runs in a matter of seconds.

3.6 Affected Processors and Configurations

We tested five different processors from the 6th generation, which introduced Intel SGX, up to the 10th, which has hardware mitigations for recent microarchitectural attacks [75]. We give the details of the CPUs tested in Table 3.2. For each processor, we tested the minimum microcode version supplied by the mainboard and the most up-to-date version as of February 2020. Each CPU was tested by computing the attack success rate for various alignments, as done in Figure 3.3. The Frontal attack was successful on all tested CPUs and microcodes.

Our measurements indicate that the processors can be separated into two groups with similar behavior: processors with and without hardware mitigations against various microarchitectural attacks. Interestingly, newer processors with hardware mitigations built-in were more susceptible to our attack, whereas older processors with mitigations in microcode seem to add noise and thus have lower success rates on average. More in-depth analysis revealed that the most recent microcodes on processors without hardware

Table 3.2: *List of all the processors we tested with their respective microcode versions. The Mitig. column indicates whether the mitigation against known microarchitectural attacks such as Spectre and Foreshadow is implemented in hardware (HW) or μcode.*

Processor	μarch	Launched	μcode	Mitig.	Vulnerable
i7-6700HQ	Skylake	Q3'15	0xc2	μcode	yes*
i7-6700HQ	Skylake	Q3'15	0xd6	μcode	yes*
i7-7700	Kaby Lake	Q1'17	0x48	-	yes
i7-7700	Kaby Lake	Q1'17	0x8e	μcode	yes*
i7-9700K	Coffee Lake R	Q4'18	0xb8	HW	yes
i7-9700K	Coffee Lake R	Q4'18	0xca	HW	yes
i9-9900KS	Coffee Lake R	Q4'19	0xb8	HW	yes
i9-9900KS	Coffee Lake R	Q4'19	0xca	HW	yes
i9-10900K	Comet Lake	Q2'20	0xca	HW	yes
Xeon E-2278G	Coffee Lake R	Q2'19	0xb8	HW	yes
Xeon E-2278G	Coffee Lake R	Q2'19	0xca	HW	yes

*Only vulnerable in some runs (see Figure 3.8)

mitigations increase the number of cycles used for `AEX` and `ERESUME` and add some randomness to our experiments. For these configurations, every run of the experiment exhibits a different behavior. Figure 3.8 shows the success rate for 500 separate runs, each with 1000 samples. Note that most of the runs with the new microcode show a random success rate. However, some runs exhibit a clear timing difference leading to a $>95\%$ success rate. The adversary can detect which behavior a particular run is going to exhibit by observing the timings of early `movs` aligned at particular addresses. Thus they could decide whether to attack or not before the secret is retrieved or provisioned and relaunch the enclave until its behavior is clearly vulnerable.

3.7 Potential Causes

The complexity of the microarchitecture of current Intel processors makes it very challenging to pinpoint the cause of the timing differences to a specific component. However, we will discuss some components which we were able to exclude decisively. We start with the memory subsystem, then we investigate the execution engines, and finally, we will focus on the frontend. For each potential culprit in these building blocks, we will describe an initial theory and then try to refute or confirm it using performance counters and other measurements. Note that the performance counters are

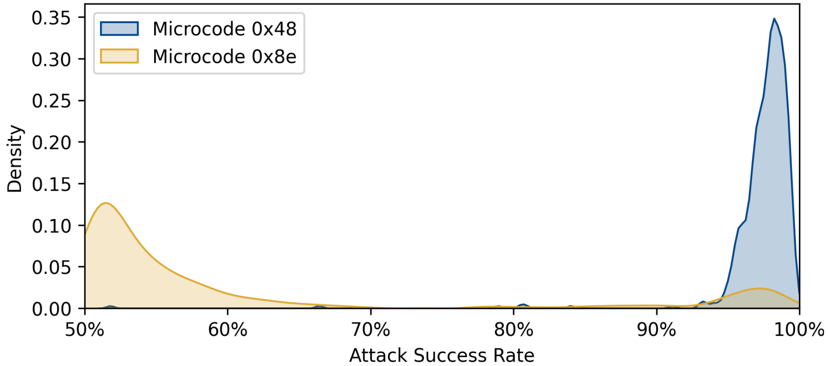


Figure 3.8: *Distribution of the attack success rate with different microcode versions of an Intel Core i7-7700 CPU – across* 500 *runs per microcode. For each run, we estimate the attack success rate as the percentage of branches the attacker guessed correctly among* 1000 *executed branches from Listing 3.2, with alignment* $X = 6, Y = 2$.

sparsely distributed over the entire core and do not exhaustively cover the entire microarchitecture. Therefore, investigating some hypotheses is very challenging if no performance counters exist for the respective part of the processor.

Memory Subsystem. Observations 3.1 and 3.2 point to potential causes in the memory subsystem. Specifically, the fact that the slow `mov` is around 100 cycles slower. For a current-generation processor, 100 cycles is a rather large delay that is usually only observed for accesses to external memory or the last level cache. However, performance counters refute any theory related to the memory subsystem since all performance counters related to external memory or last level cache did not show a difference between the slow and the fast `mov`s.

Execution Engines. The execution engine gets a list of instructions from the allocation queue as input and tries to reorder and execute them as fast as possible. As far as we know, it is completely decoupled from the frontend and does not depend on any alignment since it works on decoded micro-ops. However, given Observation 2.1, we know that the alignment influences the timing difference. We thus rule out the execution engine as the root cause of the timing differences.

Frontend. Observation 2.1 strongly hints at the frontend as the culprit since the fetch window is one of the only structures which operates at a 16 Bytes granularity, matching the 16 Bytes periodicity of the observations.

The micro-op cache is a microarchitectural structure in the frontend [76] that holds previously decoded fetch windows and serves them to, for example, repeated jumps to the same address. On a micro-op cache hit, many cycles can be saved due to not having to decode the instructions again. Our observed timing difference might stem from hits and misses in this cache. For some interrupts, the micro-op cache might miss, and the instructions must be decoded again. Whereas, for some others, it hits and immediately proceeds to the reorder buffer. However, the timing difference we observed seems excessively large for this kind of small difference in the execution path. Besides, performance counters that measure the behavior of the micro-op cache show an equivalent number of hits in the slow and the fast `movs`. Thus, we rule out the micro-op cache as a cause.

Branch prediction is responsible for predicting the future control flow. The core will fetch ahead and speculatively continue to execute in the predicted path. Branches and jumps where the target is not immediately known (e.g., the target comes from memory) both rely on the branch predictor to guess which instruction will be executed next. Hence, the resumption of the enclave could potentially suffer from a misprediction on the current enclave instruction and therefore suffer from a delay. However, all performance counters that we measured did not show any additional mispredictions for slow or fast instructions.

Summary. While we were able to decisively refute many of the most common reasons for timing differences, none of our tests were able to identify with reasonable confidence an explanation for the observed timings exploited by the attack.

3.8 Defenses

There exist various defenses against the Frontal attack, some of which we will discuss in this section. First and foremost, we want to stress that data-oblivious code [77, 64] is a principled approach that thwarts every known side- or controlled-channel attack, and as such, it also remains secure against the Frontal attack. We discuss these techniques in Appendix A.2. Nevertheless, data-oblivious code presents several challenges in practice, as it is hard to get right and results in high overhead in certain applications. Therefore, in practice, many spot defenses against known attacks have been

used since they are usually easier to apply and more performant. However, most of these spot defenses are circumvented by new attacks such as the Frontal attack. While the behavior exploited by the Frontal attack stems from the underlying hardware, the simple defense we discuss is at the software level. Hardware mitigations would also be possible, but due to the lengthy turn-around time for new processors, software defenses are more attractive.

As seen in Section 3.4, the execution time of individual instructions depends on their alignment. Particularly, branches with identical alignment do not exhibit any observable timing difference. Therefore, aligning the two branches to the same address (modulo 16) leads to indistinguishable timing distributions for both branches. We evaluated the overhead in terms of binary size and performance of this approach on three common libraries: libc, OpenSSL, and mbedTLS. We used GCC v7.5.0 with the compile flag `-falign-jumps=16` – this flag aligns all branch targets to $0x10$, thwarting our attack. The highest size overhead (3.73%) was on one of the binaries generated for libc; this, however, was the only outlier, as all the other binaries had an overhead of less than 0.5%. For comparison, compiling with `-O3` added on average 14% compared to `-O2`. To evaluate performance, we use libc-bench[7] for libc and the benchmarks that come with the libraries for mbedTLS and OpenSSL. The `strstr` test in libc-bench had the highest overhead at 30%, and libc overall had an average overhead of 1%. Depending on the evaluated cryptographic function, mbedTLS had overheads ranging from 4% to -5.5%, while OpenSSL from 3% to -4%, showing that for some cryptographic functions' implementation, the defense even provides performance boosts.

3.9 Related Work

We compare our attack and related ones in Table 3.3. In short, the main differences lie in the type of branches that are vulnerable to the various attacks. Previous defenses build either on the fact that controlled channel attacks cannot leak at sub-page granularity or that BPU attacks cannot leak the target virtual address of unconditional branches. In general, these defenses are ineffective against our attack since we exploit a fundamentally different mechanism. In the following, we describe the differences between the Frontal attack and other related attacks in more detail.

[7] https://www.etalabs.net/libc-bench.html

Table 3.3: *Overview and comparison of related SGX side-channel attacks. The first two columns indicate whether the attack can leak data-dependent or control-flow (CF) dependent secrets. The Frontal attack is the only attack that can leak the decision made for any type of branch (as long as they contain a memory store in them), even if they are based on indirect unconditional jumps (e.g., as a mitigation against BPU attacks), or if both paths are contained within the same CL (e.g., as a mitigation against cache and controlled-channel attacks).*

Attack type / Name	Data	CF	Resolution	Synchronization with Victim	Vulnerable branches
Cache [78, 15, 52, 53]	✓	✓	64 B (CL)	Interrupt / SMT / Multicore	If paths in different CLs
BPU [16, 57, 58]	✗	✓	Branch	Interrupt / SMT	If paths virtual addresses are known
TLB [79, 18]	✓	✗	4 KiB (Page)	SMT	If different data pages accessed based on path
False Dependency [80, 81]	✓	✗	4 B	SMT	If data $> 4B$ apart is accessed based on path
Port contention [82, 83]	✗	✓	μops	SMT	If paths issue different μops
PT Controlled-Channel [17, 19, 18, 84]	✓	✓	4 KiB (Page)	Page-Fault / Interrupt / SMT	If paths in different pages
Nemesis [59]	✗*	✓	Instruction type and count	Interrupt	If paths have different instructions
CopyCat [61]	✗	✓	Instruction count	Interrupt	If paths have a different instruction count
Frontal attack	✗	✓	**Instruction VA**	**Interrupt**	**Any branch (Must have a store)**

*Leaks instruction operands (if they induce different execution time). E.g., multiplication to 1 vs. multiplication with big numbers.

3.9.1 Controlled-Channel Attacks

The attacker's control over the OS enables novel noise-free deterministic side channels [17, 18, 19] known as *controlled channels* since the attacker controls the channel. Memory paging, the scheduler, and the handling of interrupts and exceptions, are a few examples of what the attacker can take advantage of – every interface between the OS and the enclaves can be leveraged in controlled channel attacks. In [17], Xu et al. modify page permissions so that the CPU generates a page fault for each page the enclave tries to access. The trace of page faults contains enough information to, e.g., let attackers reconstruct images processed in the enclave. Subsequent attacks made controlled-channel attacks stealthier by observing that the CPU sets the accessed and dirty bits [19, 18] in the page tables (PTs), thus allowing to monitor the enclave's execution without having to trigger page faults. However, the resolution of page-based controlled channel attacks is quite coarse, allowing the attacker only to know whether any access in a page (4 kB) was made but not where within it.

The coarseness of PT-based controlled-channel attacks is an element that defenses have latched onto to protect enclaves [85, 86]. These defenses either call for sensitive code to be within a page [85] or randomize the enclave's page layout so that page accesses cannot be correlated [86]. Even Intel specifies that controlled channels can be mitigated "by aligning specific code and data blocks to exist entirely within a single page" [56]. However, the resolution of controlled-channel attacks was increased through an attack exploiting legacy memory segmentation [84], which is also managed by the OS. While the attack only works under uncommon circumstances (32-bit enclaves and smaller than 1 MiB), it can observe memory accesses at 1-byte granularity.

Our attack can trace the control flow of an enclave with instruction granularity, thus increasing the resolution of PT-based controlled-channel attacks. Like other controlled-channel attacks [59, 61], the Frontal attack relies on interrupts to observe instructions and control flow within a page. However, it differs from them in the kind of branches that it can exploit. Nemesis [59] can distinguish between branches that have instructions with measurable timing differences, either because they have different kinds of instructions in their paths or because they have a different number of instructions. CopyCat [61] can track the control flow in branches with a different number of instructions. The Frontal attack allows differentiating any branch, even if both paths contain the very same instructions and are hence not vulnerable to other controlled channel attacks. The only requirement for our attack is that the branch contains at least a memory

store in it. Such higher resolution hence defeats previous defenses that rely on controlled channels being limited to observe only at a page resolution.

3.9.2 Microarchitectural Side-channel Attacks

Microarchitectural attacks exploit information leakage due to shared microarchitectural resources across different privilege domains. Among these shared resources, the ones that have been exploited the most are the cache and the branch prediction unit (BPU). We examine side-channel attacks based on these and other shared microarchitectural components below.

BPU Attacks. The BPU records the outcome of recent branches and jumps to aid the CPU speculation. As it is shared among different execution contexts running in the same core, it can leak information about the control flow of another context. The BPU was the focus of recent attacks, particularly against SGX [16, 57, 58]. BPU attacks require either SMT [58] or time multiplexing at a fine granularity between the victim and the attacker in the same physical CPU core [58, 57, 16]. These attacks are, in general, very sophisticated and require reverse-engineering of the BPU. Given how hard this is to achieve, BPU attacks are not easy to generalize to different microarchitectures and to pull off in practice [87]. These attacks are also limited to the type of branches they can exploit. For instance, they cannot leak the target virtual address of indirect jumps [16]. As these attacks give fine-grained information to the attacker, there have been a few defenses proposed against them [16, 58, 60]. Most notably, some defenses call for a holistic approach by flushing the BPU across context switches [16, 58]. Other defenses propose spot defenses, such as replacing every branch with indirect jumps [60]. BPU attacks are particularly related to the Frontal attack, as they both exploit secret-dependent branches. However, as the Frontal attack exploits a fundamentally different mechanism, any spot defense against BPU attacks is not effective against our attack.

Attacks on Caches and Other Shared Resources. Because caches are a resource shared across different execution contexts, an attacker thread can infer which accesses a victim recently made in another context by obtaining information about the cache state. While cache attacks often exploit timing variations in access latency to probe the state of the cache [88], state changes can also be detected by using instructions' side effects [89, 90]. Cache attacks target different levels of the cache hierarchy – from core-local data cache [91, 92, 93, 94, 95, 15, 52, 78, 53] and core-local instruction cache [91, 94] to the last level cache (LLC) which is shared amongst all

cores [96, 97, 98]. As code and data are shared in the upper levels of the cache (from L2), attacks that exploit them can leak both control-flow-dependent and data-dependent secrets [96, 97, 98]. Attacks on core-local caches require to be co-located with the victim and thus usually rely on simultaneous multithreading (SMT) or on accurate time-multiplexing. On the other hand, attacks that exploit the LLC can be run at the same time as the victim in another core.

The TLB is a shared buffer that stores the translation information from virtual addresses (VA) to physical addresses. It can be exploited to detect whether a victim recently accessed a data memory page [79, 18]. Since the TLB is shared only among processes in the same core, it has been exploited only using SMT so far. It can leak data accesses at a 4 kB granularity. CacheBleed [80] was the first attack to demonstrate intra-CL leakage for data accesses, achieving a resolution of 8B. It exploited cache bank conflicts and write-after-read false dependencies. Since the adversary is not in the same address space, they induce a false memory dependency by making use of 4k page aliasing – where an address x is considered the same as $x + 4096$ by the hazard detection in the processor. Cache banks are only present in older Intel architectures and, therefore, cannot be exploited on newer CPUs. Moghimi et al. [81] ported the CacheBleed attack to newer CPUs and SGX while improving the resolution to 4B in their MemJam attack. They exploit read-after-write false dependencies in the processor memory subsystem using 4k aliasing. The PortSmash [82] attack extended the resolution available to the attacker even further by being able to detect issued microops in SGX enclaves. It works by keeping specific CPU execution ports busy and monitoring their execution latency. Execution in these ports becomes slower when another context is using them, thus leaking information about their control flow to the attacker.

3.10 Conclusions

In this chapter, we observed a dependency between instructions' execution time and their alignment modulo 16. We attributed these differences to the CPU frontend and its fetch and pre-decode module. We leveraged these time dependencies to construct the *Frontal* attack, which can leak the instruction pointer of an SGX enclave at the byte-level granularity. The Frontal attack works against any kind of branch as long as it contains at least a memory write. It can attack perfectly balanced branches, even when they fit within one cacheline. We showed that the Frontal attack achieves a success rate of more than 99%, depending on the target victim code. We tested every

modern CPU microarchitecture that currently supports SGX (up to 10th gen) and found them all to be vulnerable to our attack. We demonstrated the practicality of our attack by exploiting two commonly used cryptographic libraries, mbedTLS and the Intel IPP Cryptography library. We discussed relevant defenses to the attack, such as aligning all branch targets to the same offset modulo 16. While we show that this defense has tiny size and performance overheads, we stress that, in general, secret-depending branching should be avoided to guarantee confidentially in SGX enclaves.

Chapter 4
Code Confidentiality in TEEs

4.1 Introduction

The trend of outsourcing data storage and computation has given rise to concerns about the confidentiality of not only data but also of code that is running on remote (typically cloud) services. To address these broad concerns, confidential computing, based on Trusted Execution Environments (TEEs) such as Intel SGX [10] and AMD SEV [42], has been deployed in today's commercial cloud [99, 100, 101].

TEEs allow the client to deliver their confidential code and data into a protected CPU enclave, which then isolates it from the OS and hypervisor that are running on the same machine and, more generally, from the untrusted Service Provider (SP). This is typically achieved via attestation – the client first sends the public part of its code to the SP (e.g., a VM), attests that this code is running within an enclave, establishes a secure channel (typically TLS) to the enclave, and then uses the secure channel to deliver confidential code and data into the enclave. Once the confidential code is delivered to the enclave, it can be executed in isolation. Recent years have seen an emergence of several designs that generally follow this approach, use different TEEs, and offer various trade-offs, both as academic proposals [26, 27, 28, 29, 30, 31, 32, 33] and commercial solutions [22, 23, 24, 25].

One of the core ways in which these solutions diverge is the format in which the confidential code is delivered to the enclave. They typically follow one of two approaches: *native execution* and *IR execution*, where IR stands for Intermediate Representation. We illustrate these approaches in Figure 4.1. In native execution, the developer compiles the confidential code to a *native* binary (i.e., x86) and then, after initializing a remote enclave, sends the binary to it. In IR execution, the developer compiles the confidential code to bytecode (e.g., WASM or Java) or directly sends the source code to the enclave. Whereas in the case of native execution, the enclave can simply copy the instructions from the received binary to memory and execute them, in the case of IR execution, it needs first to convert the received IR into native code. This is done by a Virtual Machine (VM)-like environment in which either a just-in-time (JIT) compiler first converts the IR code to native or an interpreter directly executes it. A number of academic and commercial systems now support either native or IR execution within TEEs.

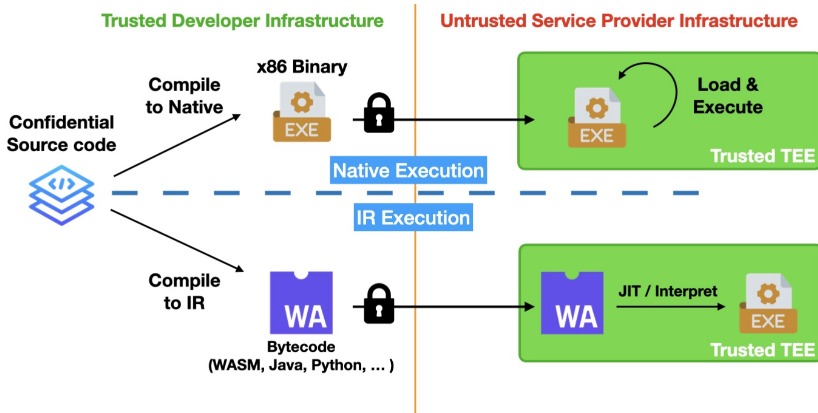


Figure 4.1: *The two main approaches providing code confidentiality with TEEs: native execution (above the dashed line) and IR execution (below the dashed line).*

WASM runtimes are particularly well supported [22, 23, 24, 30, 32] because WASM requires a small runtime resulting in a small TCB. Moreover, more than 40 programming languages can currently be compiled to WASM, with support for more underway [102].

However, even if several [26, 27, 28, 29, 31, 33, 22, 23, 25] of these systems claim to support code confidentiality for native or IR execution, so far, these claims have not been evaluated in the open literature.

This Chapter. We perform the first analysis of confidential code leakage in native and IR execution of modern TEEs. In particular, we evaluate code leakage on native execution from Intel SGX and AMD SEV TEEs (x86 ISA) and IR execution with WASM runtimes. In our evaluation, we single-step the enclaves by controlling interrupts and recording various side-channel measurements for each instruction. This allows building a trace of the execution of the victim enclave at the instruction granularity in an attempt to identify individual IR instructions or instruction sequences.

Our results show that native execution is largely robust to even the most sophisticated side-channel attacks and leaks limited information about individual instructions. IR execution, which we tested on WAMR [103], a lightweight WASM interpreter developed by the Bytecode Alliance, however, has shown to be highly vulnerable to our side-channel analysis. We successfully leaked more than 45% of the secret instructions with 100% confidence from a synthetic C program running various math

and cryptographic functions and from a chess engine written in Rust [104]. Collectively, we successfully extracted over 1 billion WASM instructions from both code sets, albeit not all with 100% confidence. This is possible because we are able to leak around 80% of the instructions in the WASM instruction set architecture (ISA) with 100% confidence. This level of confidence is obtained from just one run of the victim enclave.

These results are consistent with the expected side-channel leakage. Each IR instruction is represented by several native instructions. To identify an IR instruction, the attacker can therefore rely on a much longer side channel trace than when it tries to identify an individual native instruction. Therefore, it is clear that IR execution is generally more vulnerable to code leakage than native execution. Our results show that in the case of IR execution, such leakage is also practical, which raises questions about the security guarantees of any IR execution in TEEs.

Contributions. We summarize our contributions as follows:

- To our knowledge, this is the first study to investigate and bring forth the challenges in providing *code* confidentiality in TEEs.

- We generalize system designs aiming to provide code confidentiality in TEEs into two, native execution and IR execution, and develop a methodology to quantify and compare their code leakage.

- We analyze instruction leakage in both systems on various microarchitectures supporting TEEs from Intel and AMD. Our evaluation reveals that native execution leaks significantly less than IR execution. We also show that IR execution greatly amplifies any leakage from native execution and allows us to extract most of the confidential instructions *from a single execution*.

- To demonstrate the practicality of these findings in IR execution, we develop a practical end-to-end instruction extraction attack against WAMR, a WASM runtime running on Intel SGX. We responsibly disclosed the findings to the affected vendors (cf.Appendix B.1).

4.2 System and Attacker Model

We consider a setting in which computation is outsourced while needing to safeguard the confidentiality of the code used for computation. Two main parties are involved in this setting:

- A *Confidential Algorithm Owner* (CAO) that wants to offload computation to the cloud while keeping their code confidential; and
- A *Service Provider* (SP) that provides support for Trusted Execution Environments (TEEs).

While the SP TEEs provide memory confidentiality at runtime, the CAO cannot simply create an *enclave* (a TEE instance) containing the confidential code, ship it to the SP, and expect it to remain confidential: on both Intel SGX and AMD SEV, the initial state of the enclave is visible by the untrusted operating system (OS) and/or the hypervisor. Academic [26, 27, 28, 29, 30, 31, 32, 33] and industrial [22, 23, 24, 25] solutions address this problem by supplying the confidential code to the enclave only after the enclave has been initialized and attested. The confidential part of the code is, therefore, only communicated to the enclave after the attestation and the creation of the secure channel between the CAO and the enclave.

Typically, two main approaches are employed to supply and execute confidential code in an enclave: *native execution* and *IR execution*. Each can be further broken down into three stages: (i) compile, (ii) attest, and (iii) deploy and execute.

(i) Compile. In this stage, the CAO compiles its confidential source code for the TEE. Native execution approaches [26, 27, 28, 29, 25, 24, 33] require the CAO to compile to a native format (we focus on x86 object binaries). In IR execution [22, 23, 25, 24, 30, 31, 32], code gets compiled to an *intermediate representation* (IR) chosen as a compilation target, e.g., WebAssembly (WASM) bytecode, Javascript, Python, or Go. In some of the systems, the compilation step is skipped as the TEE directly interprets the source code.

(ii) Attest. In this stage, the CAO deploys an initial, non-confidential code with the SP and attests that this code is initialized in the enclave. Attestation ensures that the initial enclave has been deployed in a legitimate TEE and that its integrity is guaranteed. This initial enclave code is often provided by the chosen framework or SP [25, 22, 24]. As part of attestation, the CAO bootstraps a secure channel (e.g., TLS) with the enclave. On this secure channel, the CAO sends either the confidential code to the enclave or a key to decrypt a confidential code image already contained in the initial enclave.

(iii) Deploy and Execute. After the attestation, the CAO instructs the initial enclave to execute the confidential code. In native execution, this

is straightforward – the enclave simply jumps to the entry point of the x86 confidential code, which was stored in its memory as a result of the previous stage. In IR execution, the initial enclave contains an interpreter (e.g., WASM or Python), potentially with a just-in-time (JIT) compiler; the confidential instructions get interpreted, and, if a JIT compiler is available, some parts get compiled to native (x86) to speed up the execution.

4.2.1 Attacker Model

The goal of the attacker is to leak the instructions and, therefore, the confidential code that is executing in the TEE. Here we assume that the attacker is either the Service Provider (SP) or has privileged access to the server in which the confidential code is executing, i.e., the attacker controls the supervisor software, that is, the hypervisor (on a system with AMD SEV) and/or the operating system (for Intel SGX). This is a standard attacker model for TEEs [43, 11]. The attacker can see the non-confidential, initial enclave code as this code is provided in cleartext to the OS and hypervisor to load the enclave; typically, this code is public. We assume that the attacker has no control over when the confidential algorithm is executed and which secret inputs are given to it. This assumption impacts the side channels available to the attacker, as for instance, in this setting, it is difficult to i) restart an enclave a large number of times to average out noise, and ii) correlate the instructions across multiple runs – as different code paths might be executed depending on the supplied inputs.

Since the attacker has control over supervisor software on the system, they are able to: manipulate interrupts, observe changes to paging management structures (such as page table entries), and other information available to the OS, such as the last branch record (LBR). These capabilities[1] allow the attacker to single-step the TEE execution (through interrupts), see whether memory read and writes are executed (through the page tables), the approximate location (down to the cacheline) of memory read and writes, which code-page is being executed, whether some types of jumps were executed, and the execution time of interrupted instructions. We refer to an attacker with these capabilities as the *state-of-the-art* (SotA) attacker.

Throughout the chapter, unless otherwise specified, we employ a SotA attacker. However, when necessary to establish upper bounds on code leakage, we use a stronger attacker model, which we refer to as the *ideal*

[1] As demonstrated in the literature against SGX [67, 59, 19, 18, 16, 61], they apply to AMD SEV as well, as discussed in Section 4.9.

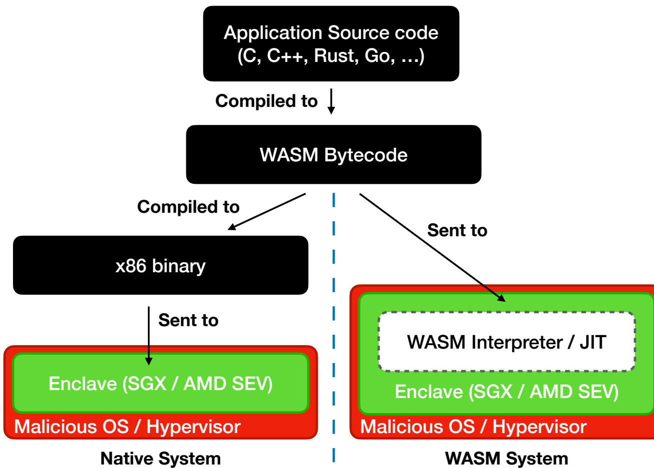


Figure 4.2: *The two approaches to code confidentiality in TEEs. The native execution enclave (left) gets the source code compiled to x86; the IR execution enclave (right) gets as input WASM bytecode. Both systems operate in an environment with a malicious OS and are tasked with executing the same source code.*

attacker. As the ideal attacker is specific to the system for which we want to estimate an upper bound, we will only introduce it when needed in the following sections.

4.3 Leakage Analysis Overview

To compare the leakage in native and IR execution, we instantiate them in two systems, the *native system* and the *WASM system*, illustrated in Figure 4.2. The *native system* accepts and executes confidential instructions in x86 (native) binary format. The *WASM system* implements IR execution by accepting as input WASM bytecode instructions. The WASM system enclave can then either interpret the bytecode or process it with a JIT compiler before execution. We refer, in general, to interpreters and JIT compilers as *translators*. We choose WebAssembly (WASM) to evaluate intermediate representation (IR) leakage due to its widespread adoption, ample language support (more than 40 languages can be compiled to WASM bytecode [102]), and the existence of multiple stable and

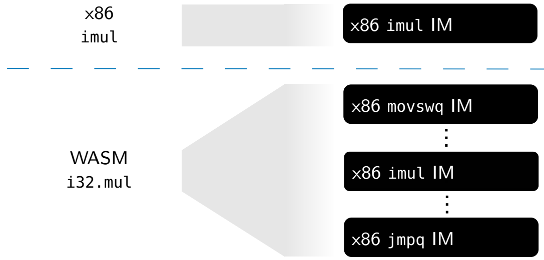


Figure 4.3: *Sample trace collection during the execution of an x86* `imul` *and one of its WASM equivalents,* `i32.mul`*. The x86 instruction generates a single Instruction Measurement (IM), while the WASM instruction generates 9 IMs due to executing 9 underlying x86 instructions.*

lightweight runtimes. Further, it can easily be compiled into native code, making the comparison between the two systems easier and more rigorous. The enclaves in the two systems get the instructions in different formats from the same source program. We compile the source code to WASM bytecode and then the bytecode to x86 outside the enclave (cf. Figure 4.2). The native system is given the final x86 binary, while the WASM system is given the intermediate WASM bytecode. Thus, the two systems are tasked with executing the very same program, allowing us to attribute any possible differences in leakage to the system running the instructions.

There are two fundamental differences between the native system and WASM system that influence their susceptibility to side channels: (i) translators often execute more low-level instructions than equivalent native binaries, and (ii) the instruction set architectures (ISAs) of native instructions are usually considerably bigger than the ISAs used for interpreted languages. Combining these two observations, our hypothesis is that the WASM system is potentially leakier than the native system due to having longer (and thus more unique) patterns of execution traces and having fewer possible instructions in the ISA that generate these traces. In the following, we expand on these differences.

Number of Executed Native Instructions. Translators of high-level languages with powerful semantics execute multiple native instructions for each high-level instruction. These translators thus *amplify* the amount of information an attacker can collect during the execution of interpreted code, compared to attacking a native system. For example, Figure 4.3 shows the difference in collected traces by an attacker when profiling one x86 instruction versus one of its equivalents in a WASM interpreter.

All translators, from high-performance JIT-based to interpreters, must perform two steps to execute a binary: first, they have to parse the code and, second, execute it. Parsing usually involves looping over each instruction of the input code, decoding it, and preparing it for execution (e.g., with a `switch-case` statement, as shown in Listing 4.1). As the underlying architecture does not provide single complex instructions to perform these operations, multiple native instructions are executed while parsing a single WASM instruction. Not only this but since different WASM instructions require different actions by the parser, the amplified instructions will differ based on which WASM instruction is being parsed. Effectively, this creates an exploitable control-flow dependency. Similar issues arise during execution. For instance, the WASM `add` instruction adds the last two values from the WASM stack and then writes the result back to the stack. An interpreter needs first to read these values and then write the result back, generally using multiple native instructions for this task. In contrast, on x86, it is possible to perform all these operations with a single `add`. In summary, the WASM system enclave executes several (and different) x86 instructions for each WASM instruction *both during parsing and execution*.

While Listing 4.1 shows the implementation of the loader for the WAMR interpreter [103], other WASM projects we inspected (Wasmtime [105] and Wasmer [106]) have similar implementations. In fact, we remark that the amplification described above with the related control-flow dependency on input instructions is likely to be found in any interpreter or compiler available today. However, different implementations will exhibit different amplification factors, as, compared to each other, they might employ a different number of x86 instructions to parse and emulate high-level instructions. This aspect is crucial as it affects the exploitability of the high-level instructions.

Difference in ISAs. The WASM Instruction Set Architecture (ISA) is significantly smaller than the x86 ISA (between ≈6x and ≈14x, depending on the x86 microarchitecture). Since the attacker knows that the enclave accepts only valid instructions, the attacker has fewer instructions to guess from in the WASM system than in the native system. To give a concrete example of why this helps the attacker, consider the `add` instruction in x86 and WASM. In the WASM case, it can only add the two most recent values in the stack, while in x86, many variations are possible, e.g., adding from different locations in memory, from registers, or even vectors. Assuming an attacker that can only leak the opcode (i.e., an `add`), this reveals more information in the WASM system than in the native system.

```c
static bool
wasm_loader_prepare_bytecode(...) {
  ...
  while (p < p_end) {
    opcode = *p++;
    emit_label(opcode);

    switch (opcode) {
      ...
      case WASM_OP_NOP:
        skip_label();
        break;

      case WASM_OP_IF:
        PRESERVE_LOCAL_FOR_BLOCK();
        POP_I32();
        ...
```

Listing 4.1: *Excerpt of the main loop of the Bytecode alliance WAMR interpreter [103] (commit `b554a9d`) responsible for loading a WASM binary. `opcode` (line 5) is the opcode of the current WASM instruction being parsed. This listing shows how a control-flow dependency on the `opcode` usually manifests (line 8) in WASM interpreters and compilers, and how different instructions exhibit different amplification factors. For instance, `WASM_OP_IF` (line 14) requires multiple operations to be translated, amplifying the information available to the attacker compared to the equivalent functionality in x86 (usually a single instruction).*

4.4 Methodology

In our study, we single-step the enclave to collect information about each executed native instruction. We refer to the information collected for each native instruction as *instruction measurement* (IM). Given the side channels available in our attacker model, each IM contains the following information about an executed instruction: the execution time, the set of accessed code pages, the set of accessed data pages, and for each of the data pages, whether the access was a memory read or write. A series of IMs forms an *execution trace* containing all the information available to the attacker. Note that the trace contains as many IMs as the x86 instructions measured. Thus, in the native system, there is one IM per confidential x86

instruction that the attacker wants to leak. On the other hand, in the WASM system, multiple IMs are collected for each confidential WASM instruction. Finally, we can only measure instructions if they are executed; hence the execution trace only contains IMs related to the executed branches and no information about non-executed code paths.

Features. It is worth noting that not all the information in an IM can be directly used to infer which instruction was executed. This is due to two reasons: first, the measurement might be too noisy, and second, it might be only related to an instruction's inputs and not to its operand. For instance, the side channel used to measure the execution time is subject to noise, and it is, therefore, generally hard to discriminate instructions based on this measurement: a memory read (`mov`) and an addition from memory (`add`) are two very different instructions (in terms of a program's logic) that produce similar timing distributions [107]. Thus, based on the timing information alone, an attacker would not be able to distinguish between the two. With respect to the second reason, knowing the data page that was accessed does not generally contain any information about the instruction type – the relevant piece of information about the instruction is that a memory access was made, not where it was made. On the other hand, knowing whether the stack was accessed does reveal information about the executed instruction type because some instructions only operate on the stack and not on other segments of memory.

Therefore, instead of using the raw numbers contained in IMs, we collect four features: the execution latency (with a resolution of 10 cycles), the type of memory access (read/write or no access), whether the instruction accessed the stack (yes or no), and whether the instruction modified the control-flow (yes or no). We arbitrarily choose a 10-cycle resolution for the attacker to over-approximate the best current attacker capabilities. To the best of our knowledge, even the most advanced attacks that leverage instruction timings show significant noise and are not even close to a resolution of 10 cycles for current TEEs [108, 59]. More details on related attacks can be found in Section 4.8. Note also that the IM does not include cache access information, despite being within the capabilities of a SotA today. We decided to exclude this information from the IM because the relevant features from this measurement (whether memory was accessed) can already be inferred from the page monitoring controlled channel, which is easier to measure and deterministic. This highlights the difference between recovering instructions compared to data: for data

inference, precise memory accesses are important, while for instruction inference, we need to extract *metadata* about the instruction.

Candidate Sets. To be able to quantitatively compare code leakage, we introduce the notion of candidate sets. The attacker forms a candidate set for each instruction they are trying to recover. Let us assume that from the IM, the attacker can deduce that the underlying confidential x86 instruction made a memory read from the stack, e.g., because the IM contains a memory read from a page assigned to the application's stack. Then the candidate set for that IM will contain instructions such as `pop`, `mov`, and `add`, as they can all read from the stack. On the other hand, it will not contain a `push`, as this instruction always *writes* to the stack. More formally, an instruction belongs to the candidate set of an IM if and only if there exists a version of that instruction that would produce a set of observations that is exactly the IM. The candidate set is useful in that it tells us that the instruction underlying an IM can only be among the ones contained in that IM candidate set. Therefore, if the set only contains one instruction, then the attacker has recovered a target instruction. In general, we can say that the smaller the candidate sets, the more information the attacker collected (i.e., the lower the entropy). The candidate set allows us to compare the leakage in the two systems in the sense that if one system tends to produce smaller candidate sets than the other, then we can say that it is leakier – and by how much. The ISA used in the target system (x86 or WASM) helps to form an initial candidate set. Since the target system can only execute valid instructions, the candidate set of an instruction with an "empty" IM contains all of the instructions of the system's ISA.

Finally, unless otherwise specified, we only report numbers for *semantically different* instructions in the candidate sets. We define semantic equivalent instructions as instructions that perform the same task but differ only in the input operand size or type (e.g., signed or unsigned). For instance, in WASM, `i32.add` is equivalent to `i64.add`, while in x86, `movq` is equivalent to `mov`. Semantically different instructions are then instructions that are not semantically equivalent. We perform this simplification because we note that generally if a candidate set contains only semantically equivalent instructions, it can be misleading to report a higher number of instructions in it.

4.5 Leakage Analysis

We now explain how to leverage IMs to build candidate sets for instructions in the native and WASM systems, and use such candidate sets

```
1       ...
2   .loop:
3       mov    -4(%rbp), %ecx   # %ecx = z
4       imul   %eax, %ecx       # %ecx = z * x
5       mov    %ecx, -4(%rbp)   # z = %ecx
6       inc    %eax             # x += 1
7       cmp    -8(%rbp), %eax   # x < y?
8       jl     .loop            # loop if true
9   .out:
10      ...
```

Listing 4.2: *A simple assembly program with a loop that, on each iteration, computes $z = z * x$. The loop iterates y times. The variable x is stored on `%eax`, y on `-8(%rbp)`, and z on `%ecx`.*

to measure how much of the confidential code leaks. For both systems, we proceed as follows:

- First, we analyze a simple program: a small loop where each iteration computes the multiplicative product of two numbers, as shown in Listing 4.2. It is composed of 6 assembly instructions, where the two numbers are multiplied in line 4. We compile this program to x86 for the native system and to WASM for the WASM system.
- Second, we discuss the IMs obtained from its execution and analyze the candidate set sizes for each instruction.
- Finally, we estimate the leakage of the system by computing candidate set sizes for all instructions in its ISA.

In the following, we first analyze the baseline native system. We start our analysis with the SotA attacker with practical capabilities (e.g., timing resolution of 10 cycles). We then expand the attacker capabilities to account for future attacks with single-cycle accuracy, functional units occupied over time, and more. We use such an unrealistically strong attacker to determine an upper bound to leakage in the native system (Section 4.5.2). Finally, we analyze the WASM system under the SotA attacker (Section 4.5.3).

4.5.1 Leakage in the Native System

We compiled the sample binary from WASM bytecode to x86 and profiled its execution to gather its IMs: Table 4.1 shows the collected features when

Table 4.1: *View of the attacker for the* `asm` *in Listing 4.2; $y = 2$. Candidate sets contain only semantically different instructions. Collected in the Skylake microarchitecture.*

Instruction	Cycles	Memory	Stack Access	Is CF?	Candidate set size
`mov`	$0-10$	R	✓	✗	545
`imul`	$0-10$	-	✗	✗	581
`mov`	$0-10$	W	✓	✗	86
`inc`	$0-10$	-	✗	✗	581
`cmp`	$0-10$	R	✓	✗	545
`jmp`	$0-10$	-	✗	✓	23
`mov`	$0-10$	R	✓	✗	545
`imul`	$0-10$	-	✗	✗	581
`mov`	$0-10$	W	✓	✗	86
`inc`	$0-10$	-	✗	✗	581
`cmp`	$0-10$	R	✓	✗	545
`jmp`	$0-10$	-	✗	✓	23

the loop is executed twice, and the number of *candidate instructions* on Skylake CPUs. We observe that, despite combining the information from several side channels, the attacker rarely gets a candidate set with fewer than 100 instructions.

In fact, this is not the case just in the example binary of Listing 4.2, but it is a consequence of the classes of instructions that can be built with the employed side channels. As there are fewer classes than there are instructions, some instructions are bound to belong to the same candidate set, thus making them indistinguishable from each other.

Full ISA. We now turn to the entire native system ISA: by computing all possible candidate sets, we can check how many instructions of the ISA have a candidate set size below a certain threshold, with the idea that the smaller the overall candidate set sizes are, the leakier a system is. Observe that each IM maps to exactly one class, and the instructions in that class form the candidate set for that IM. This means that all the possible classes are exactly all the possible candidate sets that can be observed for a system.

However, to estimate which instructions are in which class, we would need to collect an IM for all instructions (and their variations) in the x86 ISA available in SGX and SEV. Further, we would also have to do this for different microarchitectures, as these support different extensions of the x86 ISA and thus change the set of available instructions. Instead

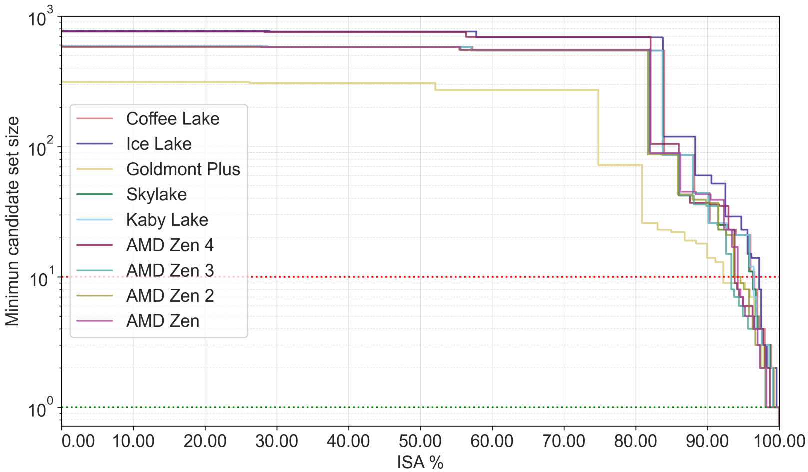


Figure 4.4: *Instruction candidate set size distribution of semantically different SGX and SEV instructions for various 64-bit x86 microarchitectures under the SotA attacker. The plot shows the minimum candidate set size that contains at least x percent of the ISA available in the TEE (SEV for AMD and SGX for Intel). This assumes the best resolution available to the SotA attacker with respect to execution time is* 10 *cycles. The dotted red line is set at $y = 10$, and it indicates that $> 90\%$ of the ISA instructions have a candidate set size greater than* 10.

of generating programs to execute all possible instructions on different microarchitectures, we adapted and reused the results of a dataset collected as part of an x86 benchmarking suite for the x86 ISA [107]. Particularly, we inferred from the dataset to which class among the ones introduced above every instruction belongs. The dataset had to be adapted to account for the fact that some instructions are illegal in SGX or that others are intercepted by the hypervisor on SEV. We describe these caveats in Appendix B.2.

We report the cumulative distribution of the sizes of the candidate sets in Figure 4.4. What can be observed from the figure is that around 80% of the instructions of the ISA belong to a candidate set containing more than 100 instructions. Note that for SEV, 1.48% of the instructions in the ISA belong to a candidate set of size 1 and can therefore be leaked to the attacker. This is due to the fact that in SEV, some instructions, such as `CPUID`, are intercepted by the hypervisor and are therefore leaked to the attacker (not through side channels, but through a system interface). There

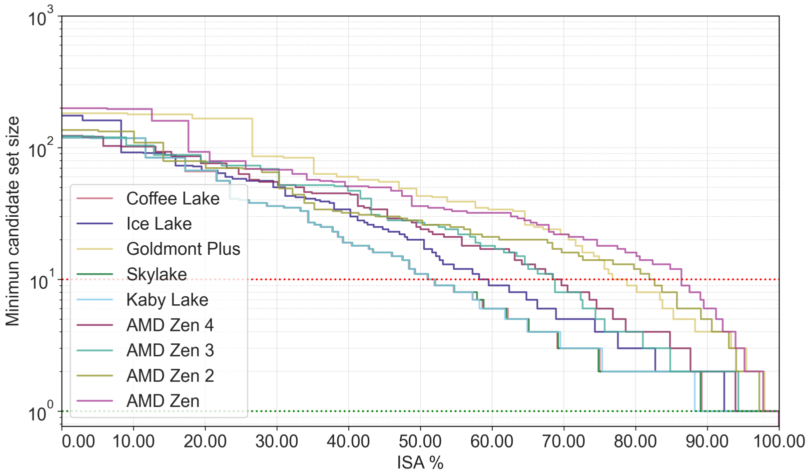


Figure 4.5: *Instruction candidate set size distribution of semantically different SGX and SEV instructions for various 64-bit x86 microarchitectures under the ideal attacker. The dotted green line is set at $y = 1$, and given where it intersects the various microarchitectures' ISA, it indicates that more than 90% of the instructions cannot be recovered even by the ideal attacker.*

are a few other instructions with a candidate set size of < 10, but they are limited to less than 8% for all analyzed microarchitectures. Thus the SotA attacker is practically never able to resolve any instruction of the x86 ISA *based on the evaluated side-channels information alone*.

4.5.2 Ideal Attacker

For the native system, we also explore different strengths of attacker models, for instance, showing how the candidate set sizes change based on different levels of cycle accuracy available to the attacker. We present these results in Appendix B.3 and discuss in Section 4.8 how these resolutions map to known attacks. Here instead, we study what we believe to be the extreme in terms of attacker strength, which we refer to as the *ideal* attacker. The ideal attacker has the capability of benchmarking instructions – like done in [107]. Note that [107] is a general method to benchmark instructions outside the enclave and hence uses capabilities currently blocked by SGX and SEV, such as reading performance counters and injecting instructions around target instructions. These capabilities also allow the attacker to observe the

utilization of individual functional units and obtain cycle-accurate execution time for each instruction. We assume that the other security properties of SGX and SEV otherwise hold, e.g., the ideal attacker cannot read the enclave memory. To the best of our knowledge, the data on single instructions collected in [107] is the most detailed and comprehensive dataset about the performance of current x86 processors to date. Since current attacks do not even get close to the resolution and wealth of information available in [107], the ideal attacker is currently far from realistic. Nonetheless, we see value in this second attacker model as it allows us to reason about a leakage model against a theoretically stronger attacker and to establish an upper bound of leakage that can be achieved.

To build the candidate sets for the ideal attacker, we construct the IM using the data in [107] as follows: cycle-accurate execution time, functional units (FUs) occupied over time, the code address accessed, the data address accessed (if any), and the type of data access (read or write). Regarding the FUs, for each instruction, we let the attacker perfectly see the order in which they are used and which other FUs could be used by the instruction. Using these very detailed IMs, we create the candidate sets by grouping together all x86 instructions for which the information in the IM is exactly the same[2]. Finally, we remove duplicate entries that are semantically similar, e.g., `mov` and `movq`. The resulting cumulative distribution of the candidate set sizes is depicted in Figure 4.5.

While the resulting candidate set sizes are significantly smaller than for the SotA attacker, around 50% of the ISA still belongs to a candidate set of at least size 10 for all analyzed microarchitectures. On the other hand, up to 10% of instructions are uniquely identifiable with a candidate set of size 1 on both SGX and SEV. Based on these results, the ideal attacker might be able to extract some instructions, but the majority of the ISA still remains ambiguous and cannot easily be leaked. Therefore, even an unrealistically strong adversary is not able to reconstruct most confidential x86 instructions from the IMs side channels. Note that this ideal attacker we considered does not have any prior information about the binary executing in the enclave. An attacker with more prior information might be able to extract more instructions even in the native execution scenario. Prior information might include information about the distributions of compiler-emitted instructions or even the distribution of instructions about key code segments (such as

[2]Exclude code and data addresses as these only contain information related to the input data and not the confidential instruction.

function epilogs). At present, the weight that prior information plays in this leakage has not been evaluated, and we leave this to future work.

4.5.3 Leakage in the WASM System

We again first consider the loop of Listing 4.2, compiled to WASM. However, while in the native system the binary only gets executed, we note that WASM translators (AOT, JIT, and pure interpreters) generally have two phases: *loading* and *interpretation*. During loading, the WASM binary is parsed, and each instruction is decoded into some internal and implementation-specific format. The second phase encompasses the execution of the loaded WASM binary.

We choose to analyze the WAMR [103] interpreter because it combines aspects of both a JIT compiler and a pure interpreter. During the loading phase, WAMR parses the WASM instructions and eliminates instructions whose results can be statically determined. For instance, the loader optimizes away instructions that load constant parameters by pre-placing their constants into the WASM stack before execution. This optimization speeds up the interpreter, as only a subset of instructions needs to be executed later. This pre-processing of instructions makes the loading phase of WAMR akin to a JIT compiler. Multiple native x86 instructions are executed for each WASM instruction during both phases. Thus, each WASM instruction of the loop of the sample program lets us collect multiple IMs: we report them in Table 4.2. In WASM, the loop is composed of 20 instructions, out of which 12 are simplified in the loading phase, leaving 8 instructions (marked in bold in the table) to be executed in the interpreter phase. In total, we recorded 1290 IMs in the loading phase of the loop and 184 IMs in the interpreter phase (with two loop iterations). Between loading and interpreting the loop, the WASM system presents a 123x increase in instructions executed compared to when the same code is executed in the native system.

Our goal is now to understand how *unique* each trace of IMs for each of these WASM instructions is. For this, we profiled each WASM instruction (see Section 4.6 for more details) and obtained their traces of IMs. With this profiling, we build candidate sets for the WASM system, considering the information obtained from multiple IMs to differentiate instructions. Table 4.2 reports the candidate set sizes for the instructions in the loop. For several instructions, the attacker gets candidate set sizes of size 1, thus perfectly recovering the instruction, which was not possible in the native system.

Table 4.2: *Attacker view of the loop in Listing 4.2 when the source code is compiled to WASM. We report the number of executed IMs recorded both when loading (as done in the first JIT phase) and interpreting each WASM instruction. Some instructions are simplified by the JIT loader and are thus not present in the interpreter trace. Instructions with a bold font are executed both in the loading and interpreting phase, while the other instructions only during loading. Numbers are computed from the same version of WAMR as in Listing 4.1. Compared to Table 4.1, here we report only one iteration of the loop (due to space constraints). The second iteration would see the bold instructions repeated. Candidate sets contain only semantically different WASM instructions.*

Instruction	# of IM per instruction		Candidate set size	
	JIT Loader	Interpreter	JIT Loader	Interpreter
`loop`	66	-	1	-
`get.local`	63	-	1	-
`get.local`	62	-	1	-
`get.local`	63	-	1	-
`i32.mul`	33	9	4	6
`i32.store`	91	14	1	1
`get.local`	63	-	1	-
`i32.load`	91	14	1	1
`set.local`	80	-	1	-
`get.local`	63	-	1	-
`i32.load`	91	14	1	1
`set.local`	80	-	1	-
`get.local`	62	-	1	-
`i32.const`	55	-	1	-
`i32.add`	33	9	4	6
`local.tee`	97	7	1	2
`get.local`	62	-	1	-
`i32.lt_s`	33	11	4	6
`br_if`	35	14	1	1
`end`	67	-	1	-

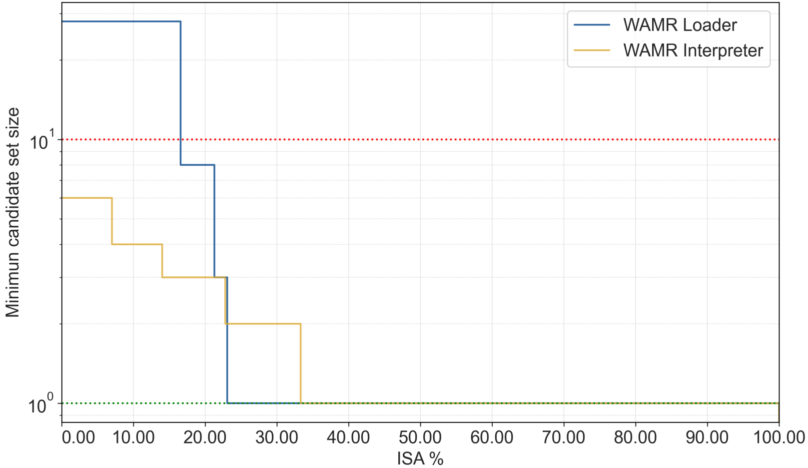


Figure 4.6: *Candidate set size distribution of WASM instructions in the WAMR interpreter under the SotA attacker. Only semantically different instructions are included in the candidate sets. The green dotted line is at* $y = 1$*, where candidate sets of that size offer no confidentiality.*

However, even in WASM, some instructions are very similar to each other, e.g., instructions that require few x86 instructions to execute tend to still be challenging to classify accurately. For instance, in the WAMR interpreter, the `i32.add` and `i32.sub` instructions are both implemented with 9 x86 instructions and differ for a single one: `i32.add` uses an x86 `add`, wherein `i32.sub` has an x86 `sub`. Since the side channels available to the SotA attacker cannot distinguish between these two instructions, `i32.add` and `i32.sub` end up in the same candidate set[3]. We can also observe this in Table 4.2: instructions with a small number of IMs tend to have bigger candidate set sizes.

In summary, the WASM system leaks more instructions of the example loop compared to the native system, with 85% of its instructions being fully leaked (compared to 0% in the native system).

Full ISA. Similarly to the native system, we compute all possible candidate sets of the WASM system: if the candidate sets tend to be small for a large percentage of the WASM ISA, then the system itself cannot provide code

[3]Interestingly, the attacker can still distinguish these two instructions because they differ in multiple instructions in the loading phase.

confidentiality, as this attack will likely extend to different WASM binaries besides our sample program.

For this, we obtained the IMs of each WASM instruction while profiling a WASM test suite [109]. The test suite we used is developed to comply with the WASM standard and ensures we reach a good coverage for all the 172 core WASM instructions. We depict the distribution of the WASM instructions' candidate sets that we obtained for WAMR in Figure 4.6. As can be seen, almost 80% of the ISA has a candidate set size ≤ 2, both in the loading and interpreting phase (which, in an actual attack, can be combined). Compare this to the native system, where even the ideal attacker could, at best, recover 10% of the ISA instructions, and it is clear that the WASM system is leakier than the native system. Finally, not only is the WASM system leakier, but the results also highlight that a SotA attacker can practically break code confidentiality for at least 70% of the WASM ISA.

4.6 IR Instruction Leakage in Practice

We now describe how to extract the confidential WASM instructions from the WASM system. We focus on SGX due to the availability of better tooling in this platform. We discuss in Section 4.9 to what extent these results extend to SEV as well.

We depict our attack in Figure 4.7. In the *Profiling Phase*, the attacker single-steps the enclave execution to collect IMs for each possible WASM instruction and generates a database of patterns. We detail this phase in Section 4.6.1. In the *Attack Phase*, the attacker again single-steps the enclave execution while the target WASM program is being interpreted. Here, the attacker obtains a single stream of IMs that need to be segmented correctly before matching each segment with the previously profiled patterns. We describe this phase in Section 4.6.2.

4.6.1 Profiling Phase

In this phase, the attacker's goal is to profile the target translator and generate patterns of traces for each WASM instruction. To do so, the attacker follows the methodology described in Section 4.4: having complete control over the enclave during this phase, the attacker can know which IMs correspond exactly to which WASM instruction. For example, we do so by saving the Instruction Pointer (IP) together with the measurements

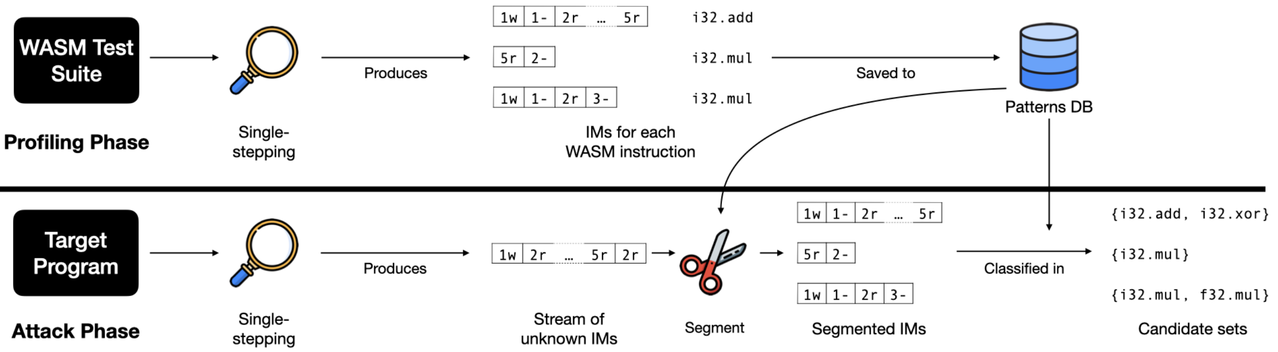


Figure 4.7: *Overview of the end-to-end attack steps.*

and obtaining the ground truth of the parsed instructions from a modified translator[4].

To build an extensive dataset for the translator, the attacker needs to profile a program that calls as many WASM instructions as possible, feeding different input data to reach good coverage[5]. We use the official WASM test suite [109], maintained by the WebAssembly Working Group that is used to test the adherence of new compilers and interpreters to the WASM specification.

On the translators that we tested, we empirically verified that we only need two pieces of information for every IM to segment an execution trace: the code page number that was accessed; and whether the x86 instruction performed a memory read, a memory write, or no memory access. Thus we represent each IM with a string composed of two parts: (i) the code page number; and (ii) the memory access type, e.g., `1r` represents an x86 instruction that was executed from page number 1 and made a memory read. Similarly, `1w` and `1-` refer to a memory write and to no memory access, respectively, from an instruction executed on page 1. When a WASM instruction is composed of multiple x86 instructions, we concatenate these symbols for the various IMs that were recorded for that WASM instruction. We refer to this string as the *pattern* for a particular WASM instruction.

Profiling the WAMR Interpreter

We now further discuss how we extract these patterns in the WAMR [103] interpreter during both of its execution phases: loading and interpreting.

Loading. During the loading phase, WAMR loops through each instruction, as shown in Listing 4.1. By manually inspecting the binary of the WAMR interpreter, we found the addresses of the first instruction of this loop and the first instruction outside of the loop. Segmenting the loader execution trace is straightforward with knowledge of the IP: we look for the loop's entry point and create a new segment every time the entry point's IP is found in the instruction trace. When we encounter the first instruction outside of the loop, we stop segment generation and restart it when we reencounter the beginning of the loop. This gives us a pattern for each

[4]The instruction pointer is not available to the attacker in the attack phase but represents valuable information: printing such ground truth of executed instructions is helpful for verification purposes, as the translator might, e.g., parse some instructions twice or skip some of them.

[5]Feeding different input values is important because the same WASM instruction might be executed by a different set of x86 instructions depending on what inputs are given to it, as we discuss in Section 4.6.2.

loop iteration. Then, to know which WASM instruction corresponds to each iteration, we modify the WAMR interpreter to record which instruction was parsed in which iteration. Note that *only* the ground truth is obtained from a modified WAMR version: the execution trace to attack is obtained from an unmodified version.

Interpreting. In the interpreter phase, the WAMR control flow is more involved than in the loading phase. The interpreter executes one instruction, then fetches the pointer of the next instruction from memory and directly jumps to it – without performing any loop. Crucially, every jump to the next WASM instruction is implemented as an indirect jump (e.g., `jmp *rax`). Thus, to segment the execution trace of the interpreter, we look for indirect jumps in the execution trace. Since we have the IP for each IM, we can check on the WAMR interpreter whether the instruction at that IP is an indirect jump. Whenever we encounter an indirect jump, we create a new segment[6]. Similarly to the loader, we need to label the segments: we again modified the WAMR interpreter to print the instruction label every time it starts interpreting a new instruction. This allows us to get the ground truth of labels for each test in the test suite. We then assign the labels to each of the segments obtained by monitoring the IP of the execution trace.

The process would follow a similar flow in other translators: what the attacker needs is a way to find instructions boundaries based on the IP (either by manual inspection or automatically) and a way to map each segment to (known) WASM instructions.

Fused Instructions Handling

So far, the way we described to build patterns for WASM instructions does not properly account for *fused instructions* from the CPU: separate x86 instructions that the CPU executes as one. When single-stepping with interrupts, these instructions will be stepped through atomically – thus, we will only encounter one IM in the execution trace instead of two. This phenomenon has also been documented in previous work [108, 61]. However, while previous work observed deterministic instruction fusion [61], we observed that for the same pair of instructions in the program (at the same virtual address), it could happen that the instructions sometimes execute unfused. This is the case even when the same input data is given to the program and both with and without hyperthreading enabled. We hypothesize that this behavior is due to the

[6] This approach only works if indirect jumps are used only at the boundary between two instructions, as is the case in the WAMR interpreter.

precise timing at which the interrupt is delivered in relation to the stage of the execution of the to-be-fused instruction pair. However, the timing at which the interrupt is delivered cannot be controlled to such precision, and therefore the behavior randomly occurs, albeit somewhat infrequently. Note that we collect significantly larger instruction traces compared to [61] (e.g., some traces we obtain have more than 1 billion instructions) and hence have a higher likelihood of observing this behavior compared to [61].

Unfortunately, this leads to the pattern of WASM instructions being non-deterministic. Theoretically, we could collect every possible variation of one WASM instruction, repeating a trace collection many times until we get all possible patterns. However, this approach is infeasible in practice for two reasons. First, since, when interrupted, the CPU non-deterministically fuses instructions, collecting all possible patterns for a WASM instruction requires many repetitions and is not guaranteed to terminate. Second, the number of different traces needed to be collected grows exponentially with the number of possible fused instruction pairs. Just having 10 fused instructions pairs in a trace requires 1024 patterns to be collected and stored.

We addressed this issue by detecting which IM could be related to fused instructions and then saving only the fused version of the pattern. Alongside the pattern, we save an array of positions that could potentially be “unfused”. This representation is not only compact (we need to save only one version of the pattern) but also allows us to match any combination of unfused instructions in the pattern efficiently. Knowing which IMs are related to fused or unfused instructions is done by cross-referencing the x86 instructions of the WAMR loader with the IP recorded for the IM.

4.6.2 Attack Phase

Trace Segmentation. In the attack phase, the adversary now targets a production enclave with the target confidential algorithm and profiles it to obtain an execution trace. As the attacker cannot obtain the IP, segmenting the different IMs for each WASM instruction is more difficult in this phase. However, by representing the full execution trace as a string, we can reduce it to the well-known string-matching problem. Segmenting the trace then proceeds as follows. Starting from the beginning of the string, we try to match all of the previously collected patterns. We then take one of the matches[7] and advance the starting pointer to just after these

[7] Multiple matches are possible because patterns overlap.

instructions. We then try to match a new pattern to this position in the string. If nothing matches, we backtrack and choose one of the previously found valid patterns. We repeat this process until the whole execution trace is perfectly segmented. We will discuss the performance of this algorithm in practice in Section 4.7.

Creating and Matching Patterns. The approach described above assumes that we can collect *every* possible pattern for each WASM instruction. Whilst the test suite patterns achieve a wide coverage, we still do not collect enough patterns to fully segment unseen binaries. In particular, while *linear* WASM instructions (WASM instructions that have no loops or branching conditions) exhibit only a single pattern, it is challenging to build every pattern for instructions with loops and branches. For instance, the WASM `clz` instruction is implemented in the interpreter with a loop that iterates once for every leading zero present in the input integer.

For cases of instructions with complex control flow, we leverage the observation that, generally, their start instructions and end instructions will be the same, no matter how complex the internal control flow is. Thus when we encounter more than one pattern for the same instruction, we automatically try to generalize its pattern. We do this by arranging the characters of the string representation in a tree where each node of the tree is one *token* (code page number and access type). We then add multiple patterns to the same tree and extract the common prefixes from it. Particularly, after the tree is assembled, we traverse it and collect every pattern found up to 2-3 splits of the tree. We found this heuristic to be quite accurate in practice. We do the same process both to find common beginning prefixes and to find end suffixes.

Between matching for common prefixes and suffixes and accounting for variable numbers of instructions due to fused instructions, we found that the most convenient way to apply the patterns was through regular expressions (*regexes*). This allowed us to use already existing and optimized matching engines, and to rapidly prototype different matching configurations. We automatically generated regexes for each possible segment while also keeping the regexes' complexity within bounds.

Segment Classification. As discussed above, trace segmentation and segment classification are inherently linked tasks. Given a correct segmentation, we already get “for free” a possible list of candidate WASM instructions for each segment: those are the instructions whose known patterns matched the segment. We call this a *candidate set*. In fact, this is how we generated the candidate sets for WASM that we discussed in

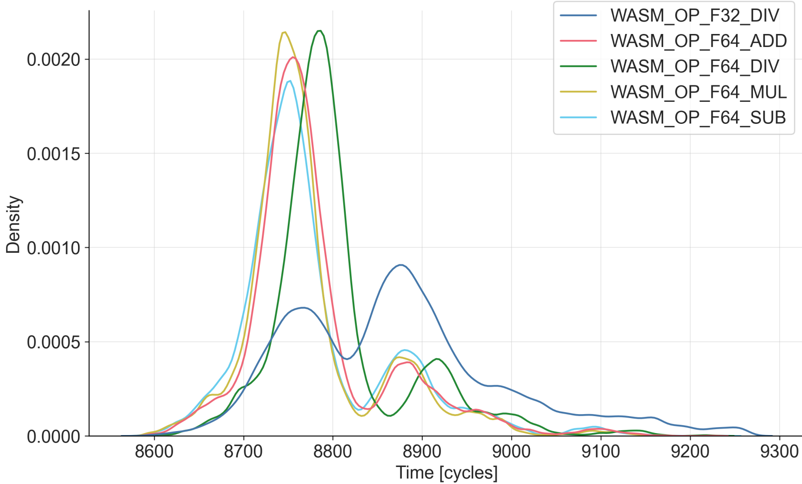


Figure 4.8: *Timing distribution of the 5th x86 instruction for the five listed WASM instructions. The two division operations seem to be following a different distribution than the others. N=11527*

Section 4.5.3. Recall that segments are generated only using the code pages and the memory access type: this information alone is so accurate to not only segment the trace but also to perfectly classify up to 80% of the WASM ISA.

Candidate Sets Pruning. We investigate whether we can further reduce the candidate set size for the remaining 20% of the WASM ISA where there is more than one candidate. In particular, IMs also contain the time spent executing individual x86 instructions, a feature that we did not use so far in our attack, as it is not fully deterministic. We explore the potential of using time measurements to prune candidate sets using a concrete candidate set obtained from the WAMR interpreter, which contains the following WASM instructions: `F32_DIV`, `F64_ADD`, `F64_DIV`, `F64_MUL`, and `F64_SUB`. Manual inspection of the interpreter's binary reveals that all of these WASM instructions are expanded into 9 x86 instructions. However, among these 9, only the fifth x86 instruction differs between the various WASM instructions. Therefore, any potential timing difference should be visible only in the 5th instruction[8]. The distribution of the recorded timings of the 5th x86

[8]We observed that surrounding instructions are also affected and exhibit timing differences, albeit smaller ones.

instruction is depicted in Figure 4.8. While the timing distributions mostly overlap, they still exhibit some differences between them.

To demonstrate the significance of these timing differences, we developed a basilar classical machine learning model that tries to classify between the aforementioned five WASM instructions using only the timing data. A simple random forest classifier [110] achieves around 45% accuracy, significantly outperforming a random guess (which has 20% accuracy). A confusion matrix is shown in Figure 4.9.

In summary, the candidate sets that we presented in Figure 4.6 could be improved by including timing information. However, the attacker would have to record multiple executions for the same confidential algorithm to establish some confidence in the results. On the other hand, the information used when segmenting is deterministic, so the attacker only needs one execution of the confidential code to build the candidate sets that were presented in Figure 4.6, and we thus deem the deterministic pipeline to be sufficient in practice.

4.7 Evaluation

We evaluated the methods and algorithms presented in Section 4.6 by using an Intel SGX enclave running the WAMR [103] runtime at commit version `b554a9d`. To collect the patterns for each WASM instruction, we single-stepped WAMR while it was executing the WASM test suite [109] (commit `e87021b`). We run only tests that do not test for exceptions, as we are interested only in correct programs, although it would be straightforward to also include these tests.

Pattern Generation. Overall, we profiled 21073 tests. Note that we single-step the test suite with the enclave in debug mode, as we need the IP to produce the segmentation patterns as discussed in Section 4.6.1. When we are profiling the WAMR loader, we only single-step the loader function (`wasm_loader_prepare_bytecode`). When we are profiling the interpreter, we single-step only the interpreter's main function (`wasm_interp_call_func_bytecode`). By monitoring the program counter after the trace collection, we observed that we can very reliably single-step the enclave through interrupts, as no instruction was skipped for any of the tests in the test suite. Hence we run each test only once. In our machine (with an Intel i9-9900KS CPU), this takes about 24 hours for the loader and about 36 hours for the interpreter. In total, we found 1576 unique patterns for the loader and 345 unique patterns for the interpreter.

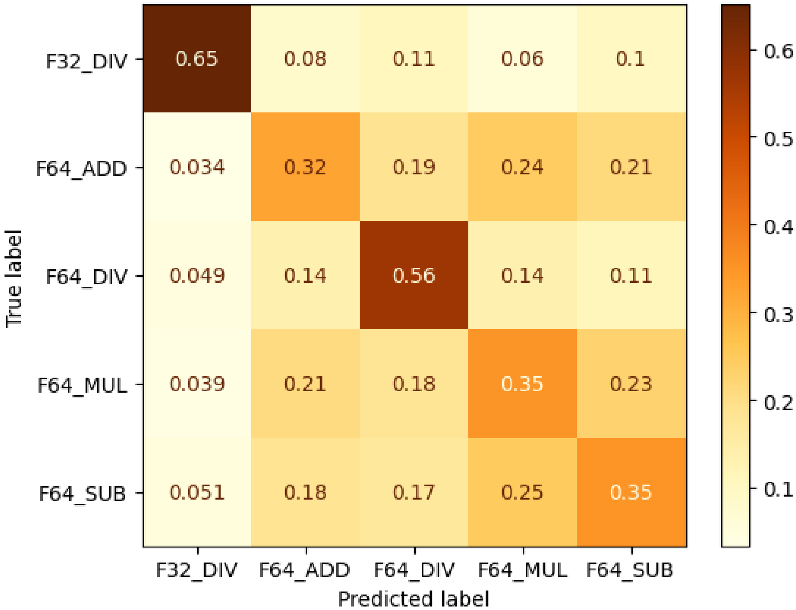


Figure 4.9: *Confusion matrix of a simple random forest classifier for five WASM instructions. The classifier is pretty confident about the two divisions but cannot distinguish the other 3 instructions.*

Using the methods described in Section 4.6.2, we then created 137 regular expressions for the loader patterns and 133 for the interpreter ones.

Instruction Matching. The SotA adversary can be instantiated in practice, and thus we performed our evaluation with real-world experiments. To test the generality and usefulness of the patterns, we used them to classify single WASM instructions in three synthetic programs. One of the programs is written in C and computes various cryptographic functions. The other two are written in Rust. One is part of a chess engine [104], while the other computes the hash of its inputs. We compiled these programs to WASM and then gave them as input to an initial enclave running WAMR. We single-stepped this enclave in production mode (i.e., without getting the IP information). Not all possible instruction patterns of these programs were present in the test suite. We verified this by naively trying to match the patterns we collected from the test suite and found that some parts of the trace could not be segmented. However, we were able to fully segment

the trace using generalized regex patterns (cf. Section 4.6.2). The C code, the Rust chess code, and the hash code respectively executed $474M$, $431M$, and $62k$ WASM instructions. When loading the code, they parsed $9k$, $38k$, and $49k$ WASM instructions, respectively. Single-stepping the interpreter phase took around 10 hours for both the chess engine and the C code and a couple of seconds for the hash engine. Single-stepping the loading phase completed in less than 5 min. Roughly the same amount of time was required to segment the traces.

From the loading phase information, we perfectly recover 46%, 49%, and 50% of all the instructions in the C code, the Rust chess engine, and the Rust hash code, respectively. At least 65% of the instructions belong to a candidate set of size ≤ 3 in these three programs. Only looking at the interpreter phase, we recover around 28% of instructions with perfect information. Note that these percentages are obtained from a single execution trace and without taking into consideration the execution time of the instructions.

Known Programs Classification. An application of the recovered WASM instruction traces is using it to classify which program or library is executing in the TEE among a fixed known set. For instance, this allows checking if a vulnerable version of a library is present in the confidential code supplied to the enclave. This is a useful building block for other attacks or could be used to check license violations.

Note that IR execution is particularly vulnerable to this classification task compared to native execution. This is because in native execution the attacker can only measure the instructions from the executed code paths. This implies that the attacker would need to either know the input of the enclave or have a trace for every possible code path of the target function/library, which is being checked for presence in the enclave. On IR execution, on the other hand, the loading phase is particularly well suited to match known segments of code. This is because, generally, instructions are parsed sequentially and in the same order across executions, no matter what other inputs are provided to the enclave. Not only this but functions are also parsed independently in the WAMR loader, allowing the attacker to even check for individual matching function signatures of a library.

We note that smaller functions are generally harder to classify than larger ones (where it is sufficient to just match with 100% confidence a couple of marker instructions in them). We thus tested several small functions by trying to match their presence in a larger library. We took

the Go Ethereum implementation[9] and compiled it into WASM. We copied the implementation of 10 individual arithmetic functions (responsible for handling big number operations) of this project and used them in smaller programs. These smaller programs simply contain a `main` function that calls the copied library functions. We then collected a trace of the loading of these small programs and segmented the WASM instructions from these traces. Finally, we tried to match the traces into a trace of the loading of the whole library. We were able to perfectly match the smaller functions in the trace of the entire Go Ethereum program, thus demonstrating that segment classification is practical in the WASM system and can help us classify which confidential WASM program is running in the enclave.

4.8 Related Work

In this section, we discuss which side-channel attacks on TEEs we build upon and how they influence the information we assume the attacker gets access to (c.f., Section 4.4). Note that generally, these side channels are developed to leak data from enclaves given the knowledge of the source code. However, in our setting, we need to adapt them to work without any prior knowledge of the source code.

Stack and Memory Access. Page table-based attacks on Intel SGX exploit the untrusted OS role in managing the page tables for enclaves [43]. The page faulting mechanism can be abused [17] to notify the attacker through page faults of enclave code and data accesses. Similarly, the access and dirty bits of the page table entries can be used to monitor read and writes [19, 18] accesses performed by the enclave. Monitoring these bits while single-stepping gives the attacker a per-instruction resolution of these values. Moreover, the attacker can also detect control-flow changes if the instruction jumps/branches to another page. Note that these attacks are completely deterministic and noise-free.

Microarchitectural Structures. Additional information can be extracted from the numerous microarchitectural details made available to the OS. While performance counters are not updated in enclave mode, their values, as measured from an attacker-controlled program, can still be influenced by the enclave execution. It is also worth mentioning that the last branch record (LBR), given knowledge of the location and target of jumps in an enclave, can be used to test for branching conditions [16]. It is feasible to extract the LBR given the knowledge of the code, but it is challenging to

[9] `https://github.com/ethereum/go-ethereum`

employ this side channel in our setting given that we do not know a priori the address of the jumping instructions in the confidential code.

Instruction Timing. Instruction timing is considerably noisier than any previously described attack. To estimate the best resolution available to the attacker, we describe how related work leaks data from enclaves despite the noisy measurements. Nemesis [59] observed that while single-stepping via interrupts, the interrupt delivery time is dependent on the instruction executed by the enclave. Usually, the attacks that leverage these timing measurements [108, 59, 83] perform multiple thousands of measurements for a single instruction to reduce the noise. We note that repeating measurements is not trivial and either requires the attacker's capability of re-running the enclave arbitrarily [59] or specific instructions before the measurement to launch a microarchitectural replay attack [83]. Even with the ability to repeat measurements, these attacks usually have a resolution of $40 - 100$ cycles.

Port Contention. The final source of information we consider is related to monitoring CPU port contention. Several attacks have demonstrated that port contention is a practical side-channel attack [82, 111]. However, they usually require repeated experiments to extract a signal from their noisy measurements. Nevertheless, we assume complete knowledge of the exact functional units used in the ideal attacker in Section 4.5.

Summary. We chose to give the SotA attacker an even better timing resolution than what is currently feasible by allowing them a 10 cycles resolution from a single run. Note that we also study an ideal attacker which, among other things, is cycle accurate and can perfectly monitor the CPU port utilization. As discussed in Section 4.7, despite these capabilities, both attacker models leak very little information from the native system. On the other hand, using only controlled-channel information is enough in the WASM system to leak the vast majority of the ISA, highlighting the magnitude of the leakage amplification between the two systems.

4.9 Discussion

Our study considered an attacker with the goal of recovering the ISA instructions of the confidential algorithm, i.e., the opcodes. These results can be used in different ways: we now discuss some possible practical attacks that leverage such data.

Reverse-Engineering Algorithms. A reverse engineer that wants to understand what the confidential algorithm does can leverage our results

on semantically equivalent instructions (see Section 4.4) to further reduce the number of candidate instructions and reconstruct the logic of the algorithm. Note that our attacker only leaks the instructions but not their *operands*. However, in a language like WASM, this is irrelevant for most instructions since their operands are implicit. For instance, an addition in WASM implicitly operates on the last two values present on the stack. Thus leaking that an addition was performed is enough to also leak the operands in this case. Note, however, that even in WASM, some instructions take constant values as parameters. These instructions can move values around on the stack based on their operand. We leave the task of leaking the operands for these instructions as future work.

4.9.1 Applicability to Other Languages

For our evaluation, we chose WASM as the language to instantiate the particular IR execution that we studied (cf. Section 4.3). However, some code confidentiality designs in TEE (e.g., Scone [25]) also support different interpreted languages, e.g., Python and NodeJS. The methods we introduced in this paper can easily be applied to analyze how much the translators of these other languages amplify the instruction leakage. As far as we are aware, their translators are not designed to provide code confidentiality, so we expect them to exhibit similar levels of leakage.

4.9.2 Applicability to SEV

To our understanding, the side-channel information and capabilities of the attacker that we use for SGX apply to SEV as well. Particularly, most of the side-channel information we use relies on manipulating interrupts and on monitoring page-level accesses. While no framework exists for SEV to conveniently replicate these functionalities, we remark that the hypervisor already performs these tasks during normal VM management. For instance, the hypervisor can schedule preemption interrupts and can tamper with page-level accesses by modifying the 2-level page translation structures. We thus conclude that the results we obtained for WASM in SGX should apply to SEV as well.

4.10 Conclusions

In this chapter, we studied two different approaches commonly used for deploying confidential code into TEEs – deploying native binaries and intermediate representation (IR) – against state-of-the-art side-channel attacks. We developed a novel methodology to analyze the side-channel

leakage of these approaches. We experimentally validated our methodology on nine modern microarchitectures and showed that IR-based confidential code deployments amplify any leakage found in native execution deployments. We showed that native execution results in limited leakage even against an ideal attacker, while next to no code confidentiality against a state-of-the-art attacker can be achieved when using WASM as an IR. Note that the attacks we performed assume that the attacker has no prior knowledge of the confidential instructions. While IR execution is already unsafe even without adding prior knowledge capabilities, we leave to future work the task of investigating whether native execution still holds enough confidentiality guarantees in this setting.

Chapter 5
Preventing Single-Stepping

The attacks introduced in Chapters 3 and 4 rely on the ability of the attacker to reliably single-step the enclave for the whole duration of the attack. This capability is a powerful primitive for the attacker, as it also enables several other attacks, e.g., the ones mentioned in Section 3.9.

Two conditions are sufficient for an attacker to be able to single-step an enclave:

1. Have an interrupt whose frequency is high-enough to fire during the time window of the first enclave instruction.

2. Detect zero-steps, that is, detect whether interrupts were sent so early that the enclave could not execute any instruction.

In SGX, the first condition is facilitated by two mechanisms. The first mechanism is that the x86 architecture guarantees that, when an interrupt arrives, only the instruction at the top of the reorder buffer (ROB) will be retired (if it does not fault), and then the interrupt will be served. This is the case even if subsequent instructions have already been completed out of order. The second relates to the number of microops of an SGX instruction. Among them, the ones related to enclave state transitions (cf. Section 2.4.1) take thousands of cycles, as discussed in Chapter 3. These two mechanisms together mean that there is a large window in which to send an interrupt which will cause to retire only the first instruction after `ERESUME` (as opposed to no enclave instruction retires or multiple of them do). The APIC timer is precise enough to satisfy this condition in Intel platforms.

Despite this large window, instructions take a variable amount of cycles to execute, e.g., due to variable memory latency. Since `ERESUME`'s execution latency is not deterministic, practically interrupts are set conservatively so that, at worst, they fire too early (zero step) but never too late. This is why the second condition defined above, being able to detect zero steps, is useful. A common technique used when single-stepping SGX to detect zero steps is to look at the accessed bit of the page table entry (PTE) of the enclave. The CPU sets the accessed bit in the PTE containing the code currently executing the enclave if and only if an instruction was fetched and retired from that page. This gives a clear signal to the attacker as the

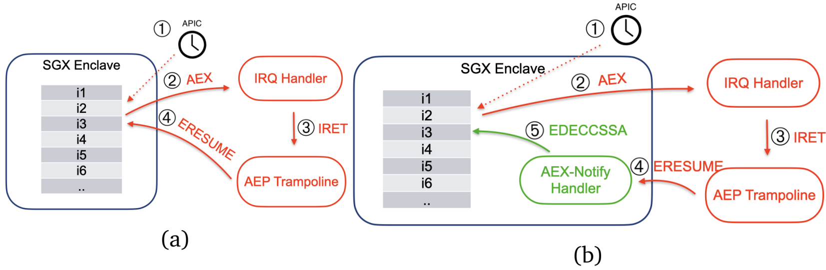


Figure 5.1: *Differences when performing (a) an AEX without AEX-Notify and (b) with AEX-Notify enabled and implemented.*

PTE is supposed to be managed by the OS, and so it can be reset in between each single-step attempt.

Note that no matter how frequent the interrupts are, they occur transparently from the point of view of the enclave, as upon receiving an `ERESUME`, the CPU restores the execution context exactly as it was when the enclave was received.

5.1 AEX-Notify

To provide enclave developers with a way to mitigate against single-stepping attacks, Intel introduced an extension to SGX named AEX-Notify [112]. In short, this extension allows enclaves to configure a handler to get called upon a return from an AEX (cf. Section 2.4.1). We depict in Figure 5.1 the process of handling an AEX both without AEX-Notify (Figure 5.1a) and with AEX-Notify enabled (Figure 5.1b). In the figure, inside the enclave, we show a list of instructions, referred to as i_x, where x is an integer. This helps us highlight how execution progresses as the enclave is being interrupted. We first recap how these steps work without AEX-Notify, and then discuss what differs with AEX-Notify enabled.

① An APIC interrupt arrives while instruction $i2$ is executing. $i2$ is at the top of the ROB when the interrupt arrives, thus all the other instructions are flushed and $i2$ gets committed. The instruction pointer (RIP) gets updated to $i3$.

② An AEX gets executed. As part of the AEX, the execution context, including the RIP, is saved in the SSA of the enclave. As part of the AEX, the instruction pointer of the AEP handler is saved in the stack

so that it can be used later to resume the enclave execution. Execution continues at the APIC timer IRQ handler.

③ The IRQ handler executes. The last instruction of the IRQ handler is the `IRET` instruction. The `IRET` instruction pops the AEP address from the stack and resumes execution from there.

④ The AEP function resumes the enclave by executing `ERESUME`. Upon executing the `ERESUME` instruction, the CPU restores the enclave execution context, including restoring the registers values from the SSA that were saved as part of AEX. The next instruction that is fetched and executed is that pointed by the `RIP` register after it has been restored from the SSA: $i3$.

To enable AEX-Notify, the enclave needs to enable some bits (set as part of `ECREATE`, see Section 2.4.1), which are reflected as part of the attestation. Additionally, an AEX-Notify handler address needs to be provided in a similar way in which enclave entry points are specified. Steps ① to ③ are identical to before, so we only describe the remaining ones next:

④ The AEP function resumes the enclave by executing `ERESUME`. `ERESUME` executes similarly to `EENTER` and calls one of the enclave entry points. No context is restored. The entry point called is a handler for the AEX event. The function starts executing and records the fact that the enclave was interrupted. If the handler decides to continue the execution, it can use a new instruction: `EDECCSSA`.

⑤ `EDECCSSA` restores the context, much in the same way `ERESUME` did in the baseline without AEX-Notify. The difference is that `EDECCSSA` is called from within the enclave to restore the context rather than being called by the untrusted application. `EDECCSSA` restores the context from the SSA, which was saved by the AEX. The next instruction that is fetched and executed is that pointed by the RIP after it has been restored from the SSA: $i3$.

As can be inferred from this description, AEX-Notify does not explicitly block either of the two conditions needed to single-step introduced at the beginning of this chapter. However, albeit we did not test this at the time of writing (due to lack of hardware supporting AEX-Notify), we speculate that it would make timing the interrupt harder as compared to just executing `ERESUME`. Thus the first condition is most likely going to be harder to achieve reliably with AEX-Notify implemented. Nonetheless, the handler needs to be

carefully implemented so that the attacker is unable to infer when `EDECCSSA` is about to be called. To give an example of what can go wrong, assume that execution in the handler crosses a page boundary just before calling `EDECCSSA`. The attacker can then monitor the access bit on the relevant PTE from another thread and start the APIC timer when it detects it.

In summary, this solution in itself does not block any of the preconditions necessary for an attacker to single-step an enclave via interrupts. Its primary purpose is to remove the transparency of single-stepping. With AEX-Notify, the enclave can be (optionally) notified every time a re-entry from an AEX occurs. Enclaves for which single-stepping is within the threat model can therefore choose to abort execution if too many AEX events are detected, something which was not (easily) possible without AEX-Notify. This solution hence, at the very least, thwarts the capability of the attacker to execute a large number of consecutive single steps, which is a *step* in the right direction. However, it potentially leaves the door open for more stealthy attacks. For instance, the attacker could single-step only a small segment of the target enclave and cover the whole execution in separate runs. Note that if this is possible, full trace collections could still be enabled but would require combining the data from multiple executions, which might not be trivial to do (cf. Appendix A.3).

Part III

Analysis of Attestation Shortcomings

Chapter 6
Relay-safe Attestation

6.1 Introduction

On TEEs such as SGX and SEV, remote attestation guarantees that the attested enclave runs the expected code. It *does not*, however, guarantee that the enclave runs on the expected computing platform. An adversary that controls the OS (or other software) on the target platform can relay incoming attestation requests to another platform. Such relay attacks are a long-standing open problem in trusted computing, as already more than a decade ago, Parno identified such attacks in the context of TPM attestation [113].

Upon first look, it might seem that relay attacks do not pose a problem for TEEs. If the attacker relays the attestation to another machine, the same security guarantees should hold since the data will only be available within the remote TEE, and the enclave code that can access the provisioned secrets is verified. However, such simple reasoning is incorrect.

In this chapter, we provide the first careful analysis of the *implications* of relay attacks on SGX and show that by relaying, the adversary increases their capabilities to attack the attested enclave significantly. One example of increased adversarial capabilities is physical side-channel attacks. If the adversary redirects the attestation to a platform that they physically control, they can mount various physical side-channel attacks, like [114, 115, 116, 117], that would not have been possible without the relay. Another example is executing enhanced digital side-channel attacks. While controlling the OS is in the SGX attacker model, it is not unrealistic that an adversary might be in the situation of controlling only user-privileged code on the target platforms. This degree of control, however, allows them to redirect attestation to another platform where they control the OS, which allows them to launch software-based side-channel attacks, such as [52, 15, 53], that leverage system privileges to attack enclaves. In Section 6.3, we explain further examples of attacks that are enabled by attestation redirection.

A typical "solution" to relay attacks is to assume *trust on first use* (TOFU). However, in many application scenarios, TOFU is neither secure nor practical. For example, solutions where attestation is performed immediately after a fresh OS installation cannot be applied to settings where OS re-installation is simply not possible. Besides, all TOFU variants

assume that the target platform OS is trusted, even if momentarily, which violates SGX's trust model.

The SGX attestation protocol is designed to be anonymous. The protocol is based on EPID group signatures [45], and thus, the remote verifier cannot distinguish whether the correct enclave on the target platform was attested or if the attestation was redirected to another platform. Upon first inspection, it may seem like relay attacks are only possible because of such anonymity features and that relaying could be easily prevented if attestation protocols were designed to be non-anonymous. However, such simple reasoning is incorrect as well. We show that all SGX attestation variants, including the "linkable" attestation mode and the Data Center Attestation Primitives (DCAP) [46], are vulnerable to relay attacks. We also explain why relay attacks would remain possible, even if all anonymity features were removed from the attestation.

Our solution. We propose a new solution, called ProximiTEE, that prevents relay attacks by leveraging a simple embedded device that is attached to the attested target platform. Our solution is best suited to scenarios where i) the deployment cost of such an embedded device is minor compared to the benefit of more secure attestation and ii) TOFU solutions are not acceptable. Attestation of servers at cloud computing platforms and setup of SGX-based permissioned blockchains are two such examples.

In ProximiTEE, the remote verifier establishes a secure connection to the embedded device whose public key it knows through standard device certification. The device performs normal SGX attestation and additionally *verifies the proximity* of the attested enclave using a simple distance-bounding protocol [118]. After the initial attestation, the device performs *periodic* distance-bounding measurements, and the communication channel created during the attestation stays active only as long as the device is connected to the same platform. Thus, the physical act of attaching the device to an SGX platform enables secure attestation (enrollment), while detaching the device will prevent further communication with the attested enclave (revocation). Neither enrollment nor revocation requires interaction with a trusted authority. This property is useful in applications like permissioned blockchains where validator nodes are separate organizations assigned by a trusted authority. The authority can issue one device per organization, and each organization is free to manage its computing resources (e.g., detach the device from one platform and attach it to another) without interaction with the authority.

Main Results. Parno [113] identified distance bounding as a candidate solution to TPM relay attacks already more than ten years ago but concluded that it could not be realized securely as the slow TPM identification operations (signatures) make a local and relayed attestation indistinguishable. Our evaluation shows that proximity verification *is* possible for SGX assuming very fast adversaries. The main reason why distance bounding protocols work for SGX but not with TPMs is that SGX is a programmable TEE where it is possible to use pre-established security associations and efficient challenge-response protocols based on simple operations such as XOR.

To evaluate PROXIMITEE, we implemented it using a USB 3.0 prototyping board. The main purpose of our evaluation is to demonstrate that the adversary cannot redirect the attestation *over the internet* to an adversary-controlled platform without being detected. We focus on such redirection, as it offers the most increased capabilities to the adversary (e.g., physical attacks). The secondary purpose of our evaluation is to determine whether proximity verification can prevent redirection to a co-located platform, like another server on the same server rack. Such relays are typically less harmful, but ideally, they should be prevented as well.

In our evaluation, we *simulate* a strong adversary that i) is only a single network hop away from the target, ii) performs the required protocol computations instantaneously, iii) has an infinitely fast hardware interface, and iv) has enabled software-based packet forwarding optimizations on the target platform. We measure the legitimate challenge-response latency on our prototype to be 185μs on average. In the case of the simulated relay attack, the average latency is about 264μs. These two latency distributions are distinguishable and allow us to set our proximity verification protocol parameters such that the adversary's probability of performing a successful relay attack is negligible (3.55×10^{-34}), while legitimate verification succeeds with a very high probability (0.999999977). Importantly, the adversary cannot increase their success probability with repeated attempts, as attestation is triggered by the trusted remote verifier. Our experiments also show that enclave revocation using periodic proximity verification is both secure and practical.

The performance overhead of proximity verification is small: the initial proximity verification adds only a small delay to the attestation protocol, and the periodic proximity verification consumes only a very minor fraction of the available USB 3.0 channel capacity. Our implementation shows that

the complexity of such a device can be small: the software TCB of our prototype is 3.8 KLoC.

Emulation Attacks. Additionally, we consider a stronger adversary that has obtained leaked, but not yet revoked, attestation keys and can *emulate* an SGX-enabled processor. Proximity verification alone cannot prevent emulation attacks, as a perfectly emulated enclave would pass any proximity test. Therefore, we propose a second attestation mechanism based on *boot-time initialization*.

In this solution, the target platform loads a small, single-purpose kernel from the attached device and launches an enclave that seals a secret key known by the device. Subsequently, when attestation is needed, the enclave can verify the proximity of other enclaves on the same platform using SGX's local attestation. This enables secure attestation regardless of potentially leaked attestation keys. Our second solution can be seen as a novel variant of the well-known TOFU principle. The main benefits over previous variants are easier adoption (e.g., no OS re-installation) and increased security (e.g., OS not trusted even temporarily).

Contributions. This chapter's contributions are organized as follows:

1. *Analysis of relay attacks.* While relay attacks have been known for more than a decade, their implications have not been fully analyzed. In Section 6.3, we provide the first such analysis and show how relaying amplifies the adversary's capabilities for attacking SGX enclaves.

2. ProximiTEE*: Addressing relay attacks.* In Section 6.4, we propose a hardened SGX attestation mechanism based on an embedded device and proximity verification to prevent relay attacks. ProximiTEE does not rely on the common TOFU assumption, and hence, our solution improves the security of previous attestation approaches. Note that the distance bounding approaches are well-known in the literature, but using such a method in the context of SGX is non-trivial.

3. *Experimental evaluation.* We implement a complete prototype of ProximiTEE and evaluate it against a very strong and fast adversary. Our evaluation in Section 6.5 is the first to show that proximity verification can be both secure and reliable for TEEs like SGX.

4. *Addressing emulation attacks.* We also propose another attestation mechanism based on boot-time initialization to prevent emulation

attacks. This mechanism, described in Section 6.6, is a novel variant of TOFU with deployment, security, and revocation benefits.

6.2 SGX Background

Intel SGX is a TEE architecture that isolates application enclaves from all other software running on the system, including the privileged OS [43]. Enclave's data is encrypted and integrity protected whenever it is moved outside the CPU chip. The untrusted OS is responsible for the enclave creation, and its initialization actions are recorded securely inside the CPU, creating a *measurement* that captures the enclave's code. Enclaves can perform local attestation, which allows one enclave to ask the CPU to generate a signed report that includes its measurement. Another enclave on the same platform can verify the validity of the report without interacting with any other external services. Enclaves can *seal* data to disk, which allows them to securely store confidential data such that only the same enclave running in the same CPU will be able to retrieve it later.

6.2.1 Remote Attestation

Remote attestation enables an external verifier to check whether a specific enclave has been correctly instantiated in an SGX-protected environment. In the following, we describe the two main classes of remote attestation supported by Intel: i) "enhanced privacy ID" (EPID) attestation [45] and ii) "data center attestation primitives" (DCAP) [46].

EPID Attestation. The EPID remote attestation is an interactive protocol between three parties: the remote verifier; the attested SGX platform; and the Intel Attestation Service (IAS), an online service operated by Intel. Each SGX platform includes a system service called *Quoting Enclave* (QE) that has exclusive access to an attestation key. The remote verifier sends a random challenge to the attested platform, which replies with a QUOTE structure, capturing the enclave's measurement from its creation, signed with the attestation key. The verifier can then send the QUOTE to the IAS that verifies its signature and correctness, checks that the attestation key has not been revoked, and in case of successful attestation, signs the QUOTE.

The attestation key used by the QE is part of a group signature scheme called EPID that supports two signature modes: random base mode and name base mode, also called "linkable" mode. Both signature modes do not uniquely identify the processor to the IAS; but only a group, like a particular processor manufacturing batch. The difference between them is

that the linkable signature mode allows to check whether two attestation requests came from the same CPU.

DCAP Attestation. Whereas the EPID attestation variant requires connectivity to an Intel-operated attestation service and is limited to pre-defined signature algorithms, the main goal of the DCAP attestation variant is to enable corporations to run their own local attestation services with freely chosen signature types. To achieve this, each SGX platform is, at the time of manufacturing, equipped with a unique *Platform Provisioning ID* (PPID) and *Provisioning Certification Key* (PCK). Intel also provides a trusted *Provisioning Certification Enclave* (PCE) that acts as a local CA and certifies custom Quoting Enclaves that can use freely-chosen attestation services and signatures.

DCAP attestation requires a trusted enrollment phase, where the enrolled SGX platform sends its PPID (in encrypted format) to a local corporate key management system that obtains a PCK certificate for the enrolled platform from an Intel-operated DCAP service. After that, the custom Quoting Enclave can create a new attestation key that is certified by the PCE enclave on the same platform. The certified attestation key can then be delivered to the corporate key management system that verifies it by using the previously obtained PCK certificate. Once such an enrollment phase is complete, the custom QE can sign attestation statements that can be verified by a local corporate attestation service without contacting Intel.

6.2.2 Side-Channel Leakage

Chapters 3 and 4, and previous work as discussed in Section 3.9, have demonstrated that the SGX architecture is susceptible to side-channel leakage. These kinds of attacks can be addressed by hardening the enclave's code, e.g., using data-oblivious coding techniques (cf. Appendix A.2).

System vulnerabilities such as Spectre [66] and Meltdown [119] allow application-level code to read the memory content of privileged processes across separation boundaries by exploiting subtle side effects of transient execution. The Foreshadow attack [120] demonstrates how to extract SGX attestation keys from processors by leveraging the Meltdown vulnerability.

Microcode Updates. During manufacturing, each SGX processor is equipped with hardware keys. When SGX software is installed on the CPU for the first time, the platform runs a provisioning protocol with Intel. In this protocol, the platform uses one of the hardware keys to demonstrate that it is a genuine Intel CPU running a specific microcode version, and it

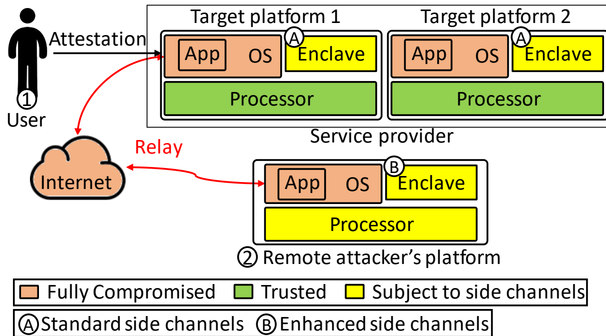


Figure 6.1: *Relay attack: The adversary redirects attestation to their own platform, which gives them increased (side channel and kernel level) abilities to attack the attested enclave.*

then joins a matching EPID group and obtains an attestation key [45] (or a signing key for the PCE enclave).

Microcode patches issued by Intel can be installed on processors that are affected by known vulnerabilities, such as the above-mentioned Foreshadow attack. When a new microcode version is installed, the processor repeats the provisioning procedure and joins a new group that corresponds to the updated microcode version, and obtains a new attestation key which allows IAS to distinguish attestation signatures that originate from patched processors from attestation signatures made by unpatched processors [45].

6.3 Relay Attack Analysis

In this section, we provide an analysis of relay attacks on SGX.

6.3.1 Relay Attacks

We consider a system model shown in Figure 6.1 that consists of three parties: the target platform, the remote verifier, and the attacker's platform. The remote verifier is a trusted party that wishes to connect and attest to a specific SGX platform. The target platform is the SGX platform to which the remote verifier intends to connect. Finally, the attacker's platform is a platform owned by the attacker that is connected to the target platform through the internet.

Adversary Model. We consider the following adversary model that we call the *relay attacker*. The relay attacker controls the OS and all other

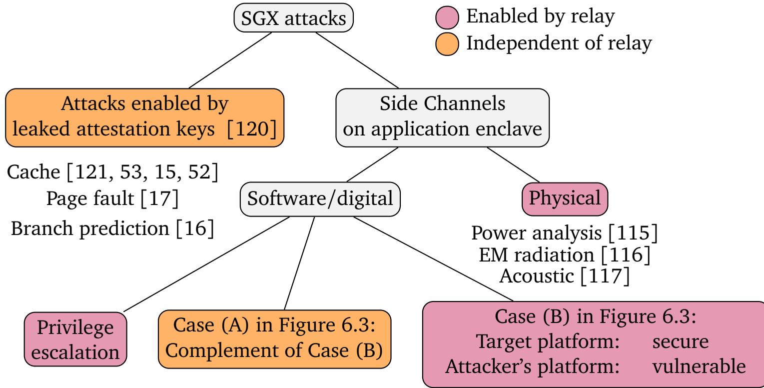


Figure 6.2: *Relay attack implications: The tree shows the types of attacks that are enabled by redirection and ones that are independent of relay.*

privileged software on the *target* platform at least *temporarily*, in particular at the time of the remote attestation. The OS compromise on the target platform may be later detected and disinfected. We consider the case in which the target platform resides in a data center or otherwise in a facility with restricted physical access. The attacker hence *does not* have physical access to the target platform (or any other co-located platform in the same facility).

The relay attacker controls the OS and all other privileged software on the attacker's platform *permanently* and has physical access to that platform. The attacker also controls the network between the target platform and their platform. At the time of the attestation, the adversary has not been able to extract attestation or sealing keys from their platform or any other SGX processor.

The Relay Attack. The relay attacker can redirect the attestation requests intended for the target platform to their platform, as shown in Figure 6.1. This is a realistic attack for two reasons. First, in the SGX attacker's model, the adversary is allowed to control the OS and can hence easily redirect any network request the target platform receives. Second, even if the attacker cannot compromise the OS in the target platform, it might be able to exploit some vulnerability of the untrusted application managing the enclave. The exploit might allow the attacker to manipulate the application's control flow to redirect attestation requests to any platform they desire.

6.3.2 Relay Attack Implications

Although relay attacks have been known for a long time [113], their implications for modern TEEs like SGX have not been carefully analyzed. Next, we perform the first such analysis.

The main consequence of attestation redirection is that it *increases the adversary's ability to attack the attested enclave* through side channels which are a well-known limitation of SGX (see Section 6.2.2). In Figure 6.2, we highlight two major classes of attacks: those that are only possible by first performing a relay attack, which we denote as "enabled by relay", and those that can be done whether or not the attacker also does a relay attack, which we call "independent of relay."

Attacks Using Leaked Attestation Keys. Our first observation is that attacks based on leaked attestation keys (e.g., ones obtained through the Foreshadow attack [120]) are independent of relaying. If the adversary has obtained a valid and non-revoked attestation key, they can emulate an SGX processor on the target platform and obtain any secrets provisioned to it.

Physical Side Channels. One major benefit of the relay, from the adversary's point of view, is that it enables *physical* side-channel attacks against application enclaves. Once a secret has been provisioned to the attacker's platform, they have as much time as they like to perform the attack. Some examples of physical side-channel attacks are acoustic, electric, and electromagnetic monitoring, which have been shown to be both effective and inexpensive means to extract secrets from modern PC platforms (see [114] for a summary of known attacks). Since the adversary does not have physical access to the target platform, such attacks are clearly not possible without relay. Hardening programs like enclaves against physical side channels is difficult and currently an open problem [114]. Therefore, developers cannot easily defend their enclaves against physical side channels that are enabled by attestation redirection.

Privilege Escalation for Digital Side Channels. Another possible benefit of relay attacks is that they may enable *privilege escalation*. In cases where the adversary has only compromised the user-space application that manages the enclave and not the OS, the application can redirect the attestation to the attacker's remote platform, where they control the OS as well. In such cases, the relay enables *digital* side-channel attacks that require system privileges. Several such attacks have been demonstrated against SGX [52, 15, 53].

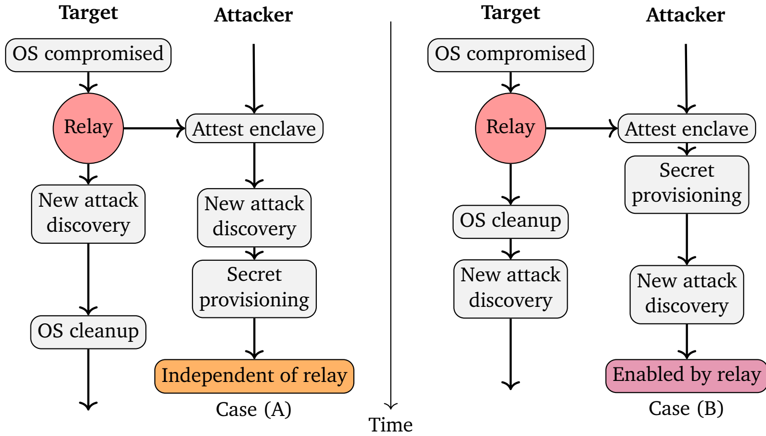


Figure 6.3: *Example sequences of events. In Case A, the attack success is independent of relay. In Case B, attestation redirection enables the attack.*

Attacks that Depend on the Timing of Events. The third, and perhaps the most subtle, implication of relay is that it can also enable software-based side-channel attacks that would not be possible to launch on the target platform due to the *timing of certain events*. These events include, but are not restricted to, the provisioning of secrets to the enclave, the possible disinfection of the target platform from malicious software, the discovery of a new side-channel attack, and even the reduction of noise due to other workloads.

We group the relative ordering of these events into two cases: A and B. Case A covers event sequences that only lead to attacks that are independent of relay, and Case B covers event sequences in which redirection gives extra capabilities to the adversary. Below, and in Figure 6.3, we provide examples of sequences belonging to these two cases:

Case A: Independent of Relay. A digital side channel is independent of relay if the adversary could perform it on the target platform as well. An example of such a case is shown in the timeline depicted in Figure 6.3, where a new attack is discovered after secret provisioning but before the target platform OS is disinfected.

Case B: Attack Enabled by Relay. Case B is reached whenever it occurs that by using a side channel, the enclave is exploitable on the attacker's platform but not on the target platform. A timeline of such a case is shown in Figure 6.3, where at the time of attestation and secret provisioning, the

enclave is hardened against all known digital side-channel attacks (using tools like Raccoon [77], ZeroTrace [122] or Obfscuro [29]). After secret provisioning, the OS compromise is detected and cleaned. Later, a new side-channel attack vector (that is not prevented by the tools used) is discovered. If the adversary performed a redirection and the secret was provisioned to the attacker's machine, the new side channel is exploitable. Without the relay, the attack is not possible.

6.3.3 Limitations of Known Solutions

Next, we review commonly suggested solutions and their limitations.

Trust on First Use. A common "solution" in the research literature is to rely on *trust on first use* (TOFU) [123]. Simple TOFU solutions assume that the OS is clean at the time of attestation or perform attestation only immediately after fresh OS installation. Both of these approaches have obvious security and deployment problems. OS re-installation is not always possible, and trusting the OS, even if momentarily, is undesirable (and violates SGX's trust model).

SGX Attestation Variants. As we explain in Section 6.2.1, SGX supports different variants of remote attestation. Unfortunately, none of these schemes prevents relay attacks without some form of TOFU assumption.

1. *The unlinkable EPID attestation scheme* is based on group signatures, and thus the remote verifier cannot distinguish between attestation responses that are received from the expected target platform or the adversary's platform.

2. *The linkable EPID attestation mode* allows the remote verifier to check if they have attested the same platform before, but the first attestation protocol run is vulnerable to relay attacks, and therefore, the remote verifier must assume TOFU.

3. *The DCAP scheme* allows corporations to operate their own local attestation services after an enrollment phase. However, if the adversary controls the target platform during the enrollment, they can replace the enrolled platform identifier PPID with the identifier of their own platform PPID' and enroll the adversary's platform instead. Thus, also the DCAP variant scheme requires trust on first use. In addition, the entire corporate key management system must be trusted at enrollment time (and after it).

Non-Anonymous Attestation. Because SGX's attestation protocol support anonymity features, like the EPID signature scheme, one may think that relay attacks are caused by such privacy protection mechanism. However, such reasoning is incorrect. Even if all anonymity features were removed from attestation, the problem of relay attacks would still persist. The root cause of relay attacks is that certified keys can be securely installed to processors at the time of manufacturing, but the processor ownership by private individuals or companies is established much later. Therefore, common PKI mechanisms do not eliminate relay attacks – unless the processor manufacturing and distribution model is completely changed such that factories start to manufacture and certify customer-specific processor batches on demand (which would be very expensive).

Other TOFU Variants. Research papers tend to use slightly different TOFU variants. For example, the ROTE system [124] assumes fresh OS installation at system initialization time, and for each used platform, it requires a local administrator to input a credential to the enclaves. As another example, in the VC3 system [26], enclaves generate a public/private key pair at the time of trusted initialization, output the public key and seal the private key. The public key can be sent to a trusted authority for certification, which then enables clients to securely connect to enclaves. Both of these solutions essentially avoid insecure attestation by pre-authorizing known enclaves during a setup phase that is assumed trusted.

In general, TOFU solutions suffer from the following limitations:

1. *OS re-installation:* Forcing users or administrators to re-install the OS is not always possible.

2. *Manual configuration:* Manual interaction tasks, such as an administrator that needs to enter credentials to enclaves during initialization, complicate platform enrollment, especially in scenarios like data centers with many enrolled platforms.

3. *Pre-defined enclaves:* Solutions that only work with enclaves that are known at the time of initialization are not applicable to scenarios like cloud computing platforms where users need to install new enclaves after platform installation.

4. *Large temporary TCB:* Modern operating systems have a large TCB, and trusting the OS even temporarily is unideal.

5. *Online authorities:* Solutions where a trusted authority needs to either certify or revoke new enclaves typically require that the authorities are online, which increases their attack surface.

6.4 PROXIMITEE

Our goal is to design a solution that addresses the above limitations of previous solutions. In short, our solution should be *secure* (no TOFU assumption, small TCB, no online authorities) and *easy to deploy* (no OS re-installation, manual configuration, or pre-defined enclaves). In this section, we provide an overview of our approach, outline possible use cases, describe our solution in detail, and analyze its security.

6.4.1 Approach Overview

We propose a hardened SGX attestation scheme, called PROXIMITEE, based on a simple embedded device that we call PROXIMIKEY. The embedded device is attached to the target platform over a local communication interface such as USB.

Our main idea is to use the combination of such a trusted device and *proximity verification* to prevent relay attacks. In our solution, the PROXIMIKEY device verifies the proximity of the attested enclave, and after successful proximity verification, it facilitates the creation of a secure channel between the remote verifier and the attested enclave.

After the initial attestation, the device periodically checks proximity to the attested enclave. The established secure channel is contingent on the physical presence of the embedded device on the target machine, and it stays active only as long as the device is plugged in. The act of detaching the device automatically revokes the attested platform without any interaction with a trusted authority. Thus, our solution enables secure *offline* enrollment and revocation.

To use our solution, enclave developers use a simple API that facilitates communications between the enclave and the device.

Security Assumptions. In our solution, the PROXIMIKEY device is a trusted component. We deem this choice reasonable since it implements only the strictly necessary functions, and therefore, it has a significantly smaller software TCB, attack surface, and complexity compared to a general-purpose commodity OS. We assume that its issuer certifies each embedded device prior to its deployment, and such certification can take place fully offline.

Concerning the security of the ProximiKey device, we employ the same adversary model introduced in Section 6.3 for enclaves. While the user's device and its private keys are never exposed to the attacker, another similar device can be in the physical possession of the attacker, which has as much time as they want to fully compromise it (run arbitrary code and extract keys).

6.4.2 Example Use Cases

Our solution is targeted to scenarios where the benefits of more secure attestation outweigh the deployment cost of a simple embedded device. Here, we outline three example cases.

Datacenter. In our first example, we consider a cloud platform provider that attaches ProximiKey to a server in a specific data center and makes the public key of the connected device known to the users of the service. Our approach is particularly well suited to cloud computing models where customers rent dedicated computing resources like entire servers. In such a setting, our solution ensures that the cloud platform customer outsources data and computation to a server that resides in a specified location. Enforcing location may be desirable to meet increasing data protection regulation that defines how and where data can be stored, even if protected by TEEs such as SGX. Revocation (e.g., when a server is relocated to another data center or function) can be realized by merely detaching ProximiKey.

Permissioned Blockchain. Our second case is a setting in which a trusted authority initializes a set of validator nodes for a permissioned and SGX-hardened blockchain. The trusted authority issues one ProximiKey for each organization that operates one of the validator nodes, which allows secure attestation of the validator platforms. Organizations are free to upgrade their computing platforms by attaching the ProximiKey to a new platform which automatically revokes the old platform without the need to interact with a trusted authority. Furthermore, since ProximiKey can only be active on one platform at a time, such a deployment enables the authority to control the identities used in (Byzantine) blockchain consensus processes.

HSM-Protected Keys. Our last case is the management of HSM-protected keys from an attested enclave. Such deployment enables the secure and flexible realization of various access control policies implemented as attested enclaves. ProximiTEE guarantees that only an enclave in the proximity of the HSM can control its keys. Such a solution provides a high

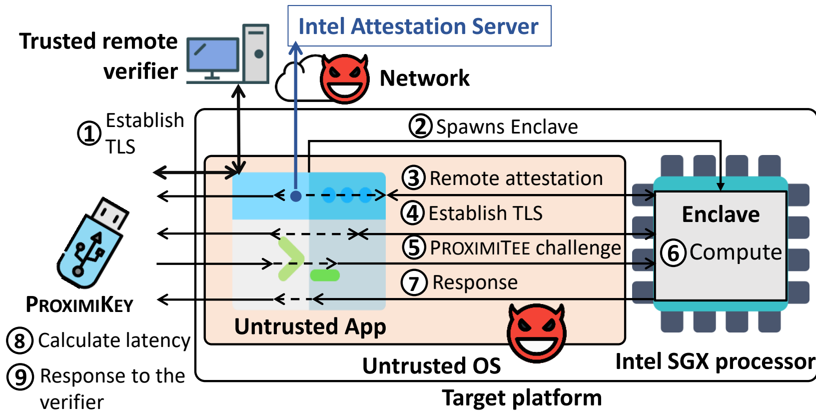


Figure 6.4: PROXIMITEE *attestation: The remote verifier establishes a secure channel to the* PROXIMIKEY *device that first attests the enclave and then verifies its proximity.*

level of protection because, at no point in time, the HSM keys are directly accessible by the enclave (which may be vulnerable to side-channel attacks) or by the untrusted OS.

6.4.3 Solution Details

Now, we explain the PROXIMITEE attestation mechanism in detail.

I. Attestation protocol. Figure 6.4 illustrates the attestation protocol that proceeds as follows:

① The remote verifier establishes a secure channel (e.g., TLS) to the certified PROXIMIKEY. An assisting but untrusted user-space application facilitates the connection on the target platform, acting as a transport channel between the remote verifier and the PROXIMIKEY (and later also the enclave). As part of this first step, the remote verifier specifies which enclave should be executed.

② The untrusted application creates and starts the attestation target enclave.

③ PROXIMIKEY performs the standard remote attestation to verify the code configuration of the enclave with the help of the IAS server or using a custom DCAP procedure (see Section 6.2). In the attestation protocol, the device learns the public key of the attested enclave.

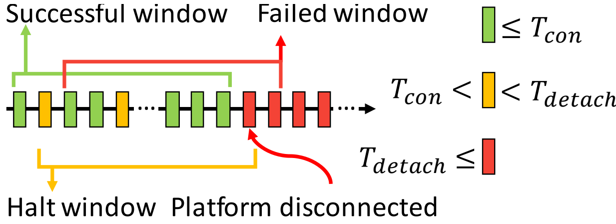


Figure 6.5: *Sliding window: for periodic proximity verification with three different types of challenge-response latencies.*

④ ProximiKey establishes a secure channel (e.g., TLS) to the enclave using that public key.

⑤ ProximiKey performs a distance-bounding protocol that consists of n rounds, where each round is formed by steps ⑤ to ⑧. At the beginning of each round, ProximiKey generates a random challenge r and sends it to the enclave over the TLS channel.

⑥ The enclave increments the received challenge by one $(r+1)$.

⑦ The enclave sends a response $(r+1)$ back to the ProximiKey over the TLS channel.

⑧ ProximiKey verifies that the response value is as expected (i.e., $r+1$) and checks if the latency of the response is below a threshold (T_{con}). Successful proximity verification requires that the latency is below the threshold for at least $k \times n$ responses, where $k \in (0,1]$ is a percentage of the total number of responses n.

⑨ If proximity verification is successful, ProximiKey notifies the remote verifier over the TLS channel (constructed in step ①). The verifier starts using the ProximiKey TLS channel to send messages to the enclave.

II. Periodic Proximity Verification. After the initial connection establishment, the ProximiKey device performs *periodic* proximity verification on the attested enclave. ProximiKey sends a new random challenge r at frequency f, verifies the correctness of the received response, and measures its latency. The latest w latencies are stored in a sliding window data structure, as shown in Figure 6.5.

As elaborated in Section 6.5, there are three types of latencies in the presence of relay attacks. The first type of response is received faster than the threshold T_{con} (green in Figure 6.5); these responses can only be produced if no attack is taking place. In the second type of response, the latency exceeds T_{con}, but it is below another, higher threshold T_{detach} (yellow); these are sometimes observed during legitimate connections and sometimes during relay attacks. And third, the latency is equal to or exceeds T_{detach} (red); these latencies are only observed while a relay attack is being performed. Given such a sliding window of periodic challenge-response latencies, we define the following rules for halting or terminating the connection:

- *Successful window: no action.* If at least k responses have latency $\leq T_{con}$ and none of the responses have latency$\geq T_{detach}$, the current window is legitimate, and PROXIMIKEY keeps the connection active.
- *Halt window: prevent communication.* If one of the responses has latency $\geq T_{detach}$, we consider the current window a "halt window," and PROXIMIKEY stops forwarding data to the enclave until the current window is legitimate again.
- *Failed window: terminate channel.* If two or more responses have latencies $\geq T_{detach}$, we consider the current window a "failed window," and PROXIMIKEY terminates the communication and thus revokes the attested platform.

6.4.4 Security Analysis

Attestation security. To analyze the security of our hardened attestation mechanism, we must first define successful attestation. We say that the attestation is successful when the remote verifier establishes a connection to the correct enclave that i) has the expected code measurement and ii) runs on the computing platform to which the PROXIMIKEY device is attached.

The task of establishing a secure channel to the correct enclave can be broken into two subtasks. The first subtask is to establish a secure channel to the correct PROXIMIKEY device. This is achieved using standard device certification. We assume that the adversary cannot compromise the specific PROXIMIKEY used. If the adversary manages to extract keys from other PROXIMIKEY devices, they cannot trick the remote verifier into connecting to a wrong enclave, as the remote verifier will only communicate with a pre-defined embedded device.

The second subtask is to establish a secure connection from PROXIMIKEY to the correct enclave. For this, we use proximity verification.

PROXIMIKEY verifies the proximity of the attested enclave through steps ⑤ to ⑧ of the protocol. These steps essentially check two things. First, through step ⑦, whether the messages are received from the correct enclave. This verification is performed by checking the correctness of the decrypted message, and it relies on the assumption that the attacker cannot break the underlying encryption – hence only the enclave that has access to the key that was bound to the attestation could have produced a valid reply. Second, through step ⑧, whether the PROXIMIKEY and the enclave are in each other's proximity. This check relies on the assumption that a reply from a remote enclave will take more time to reach the PROXIMIKEY than a reply from the local enclave.

We evaluate the second aspect experimentally. In particular, we simulate a powerful relay-attack adversary that is connected to the target platform with a fast network connection. To consider the best case for the adversary, we make several assumptions in their favor. For example, we assume that they can instantly perform all computations needed to participate in the proximity verification protocol. However, they cannot break cryptographic hardness assumptions. We define the adversary's success as the event in which proximity verification succeeds with an enclave that resides on the attacker's platform, and we denote the probability of such an event P_{adv}. We define a legitimate success as the event in which proximity verification succeeds with an enclave that resides in the target platform and denote its probability P_{legit}. In Section 6.5, we show that it is possible to find parameters ($n = 50$, $k = 0.3$, and $T_{con} = 186\mu s$) that make proximity verification very secure ($P_{adv} = 3.55 \times 10^{-34}$) and reliable ($P_{legit} = 0.999999977$).

Revocation Security. To analyze the security of the periodic proximity verification which we use for platform revocation, we must first define what it means for the attacker to break the periodic proximity verification. The purpose of the periodic proximity verification is to prevent cases where the user detaches the PROXIMIKEY device from the attested target platform and attaches it to another SGX platform before the previously established connection is terminated. Since we consider an adversary who does not have physical access to the target platform (recall Section 6.3.1), we focus on benign users and exclude scenarios where the PROXIMIKEY would be connected to multiple SGX platforms with custom wiring or rapidly and repeatedly plugged in and out of two SGX platforms.

We define the periodic proximity verification as broken if the adversary can manage to keep the previously established connection alive within a

"short delay" after the ProximiKey was detached from the attested target platform. For most practical purposes, we consider a delay of 10 ms as sufficiently short. We denote the adversary's success probability in breaking the periodic proximity verification as P'_{adv}. A false positive for periodic attestation is the event where the connection to the legitimate enclave is terminated, and the attested platform is revoked despite the ProximiKey being connected to the target platform. We denote the probability that this happens during a "long period" as P'_{fp}. We consider an example period of 10 years sufficiently long for most practical deployments.

In Section 6.5, we experimentally show that revocation can be secure ($P'_{avd} = 3.55 \times 10^{-34}$) and reliable ($P'_{fp} = 1.6 \times 10^{-4}$) while consuming only a minor fraction of the available channel capacity.

6.5 Experimental Evaluation

In this section, we describe our implementation and evaluation.

6.5.1 Implementation

We implemented a complete prototype of the ProximiTEE system. Our implementation consists of two components: i) the ProximiKey embedded device prototype and ii) the ProximiTEE enclave API which enables any application enclaves to communicate with the ProximiKey device and execute the proximity verification protocols.

ProximiKey. Our embedded device prototype is based on Cypress EZ-USB FX3 USB 3.0 prototyping board that is equipped with a 32-bit 200 MHz ARM9 core. The board communicates with the target platform over a native USB 3.0 connection that provides up to 5 Gbps of bandwidth. FX3 provides direct memory access (DMA) out of the box through its API for efficient communication with the connected platform. We use the ARM mbedTLS [62] cryptographic library for the TLS. The limited set of cipher suites in our implementation uses 128-bit AES (CTR mode) for encryption, AES-HMAC as the message authentication code, Curve25519 for Diffie-Hellman key exchange, and SHA256 as the hash function. Our prototype implementation is approximately 200 lines of code, and the code size of the TLS library is around 3.6 KLoC.

ProximiTEE Enclave API. The ProximiTEE API for application-specific enclaves is written in C++ using the Intel SGX API. The API uses the native SGX crypto library for the TLS implementation, and it is around 200 lines of code.

6.5.2 Evaluation Focus: Internet Relay

For the purposes of our evaluation, we make the distinction between two types of relay attacks. In the first type, the adversary redirects the attestation *over the internet* to another platform that is under their physical control and, therefore, in a *different location*. As we explained in Section 6.3.2, such relay attack amplifies the adversary's capabilities the most, as they can now attack the attested enclave using physical side channels, they have unlimited time to launch digital side channels, or they can wait for the discovery of new attack vectors.

In the second type of relay attack, the adversary redirects the attestation to another *co-located platform*, like another server on the same server rack. In most cases, attestation relay to a co-located platform does not improve the adversary's chances of attacking the enclave because, typically, the adversary has similar control over the co-located platform. The only exception is privilege escalation in cases where the adversary has user privileged on the target platform and system privileges on the co-located platform.

Next, we focus on demonstrating that an inexpensive ProximiTEE prototype can be configured to prevent the first (and typically more dangerous) type of relay attacks with very strong security and robustness. Later, in Section 6.5.8, we discuss the second type of relay.

6.5.3 Experimental Setup

To demonstrate that ProximiTEE prevents relay attacks (over the internet), we performed two types of experiments. First, we tested the legitimate attestation execution with ProximiTEE and measured the challenge-response latencies between our prototype and the target platform. Second, we *simulated* a relay attack, where the adversary redirects the attestation to another platform.

Assumptions and Optimizations. To consider the best possible case for the adversary, we made several generous assumptions in their favor when designing our experimental setup and post-processing our measurements:

1. *Single network hop.* Since we do not want to make any assumptions about the precise network path that the relayed attestation needs to travel, we connected the adversary's platform to the target platform via a direct 1-meter Ethernet cable, as seen in Figure 6.6. With such a setup, our goal is to simulate the most direct connectivity and the best possible latency that the adversary could achieve in relay

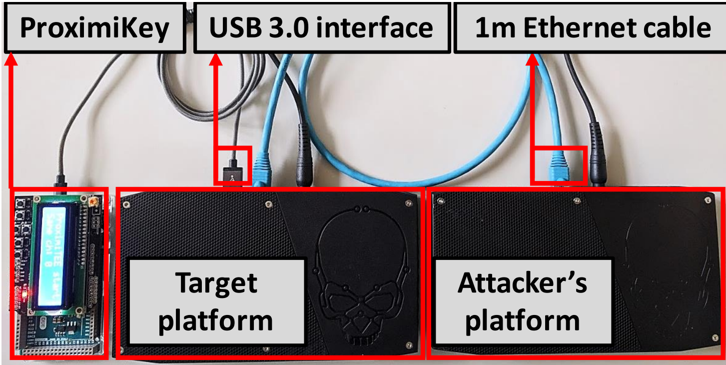


Figure 6.6: *Our experimental setup: consists of the* PROXIMIKEY *device prototype, the target platform, the attacker's platform, and the connection interfaces between them.*

attacks that take place over the internet. In most realistic attacks, the adversary would need to relay the attestation over multiple network hops, which increases the round-trip latency significantly.

2. *Instant protocol computation.* Since the adversary might have a faster processor on their platform than the one we used in our experiments, we simulated an adversary who is able to perform all computations needed for the proximity verification protocol instantly. Instant replies were simulated by fixing the randomness for the challenges and having precomputed responses for that randomness on the attacker's machine.

3. *Packet forwarding optimizations.* Since the adversary controls the OS on the target platform, they can perform software-based optimizations to reduce the packet forwarding delay. We experimented with several such optimizations. First, we tested the standard `ping` tool, which gave a latency of around 380 μs for a one-meter Ethernet connection. After that, we used the `ping` tool in so-called flood mode and measured a reduced average network latency of around 153 μs (command `ping -s 300 -af`). Flood mode achieves faster round-trip time as it forces the OS to fill up the network queue of the kernel. Based on these measurements, we chose to simulate an attacker that fills the kernel's network queues (on both platforms) similar to the flood mode to minimize latency.

We also tested other possible OS-level optimizations but did not observe any material reduction in measured latencies, and thus, in our experiments, we only use the kernel queue filling.

4. *Infinitely fast network interface.* Since the adversary's platform might have a faster network interface hardware than the one used in our experiments, we chose to simulate an adversary that has an infinitely fast network interface. In our experimental setup, both the target platform and the adversary's platform have identical network interfaces. We assume (in favor of the adversary) that the transmission time spent on the wire is negligible, and most of the round-trip latency is due to processing in the network interface. This allows us to simulate an adversary with an infinitely fast network interface by first performing latency measurements and then, in a post-processing phase, cutting down all the measured latencies by half. Note that the target platform's network interface cannot be replaced by the attacker as they do not have physical access to it.

Experiments. We conducted our experiments on three SGX platforms: two Intel NUC NUC6i7KYK mini-PCs and one Dell Latitude laptop, all equipped with SGX-enabled Skylake core i7 processors and Ubuntu 16.04 LTS installed on them. To measure latencies, we used FX-3's GPIO pins that provide 100 nanosecond level accuracy. We performed a total of 20 million rounds of the protocol for normal attestations and simulated attacks and measured the challenge-response latencies for each. We measure all of them inside the EZ-USB FX3 code. For cross-validation, we tested the ProximiKey with a high-precision oscilloscope (8 Ghz Keysight Infinium) and witnessed identical timing patterns.

6.5.4 Latency Distributions

The histogram in Figure 6.7 on the left represents the challenge-response latencies in the legitimate proximity verification. The histogram on the right shows latencies in a simulated attack (including a post-processing phase where we reduce the adversary's measured network latencies to half to accommodate the assumption of the attacker's infinitely fast network interface).

As can be seen from Figure 6.7, the vast majority of the benign challenge-response latencies take from 145 to 250μs (average 185μs, 95% of samples are in between 150μs and 200μs). The vast majority of the round-trip times in the simulated attack take from 200 to 750 μs (average 264μs, 95% of

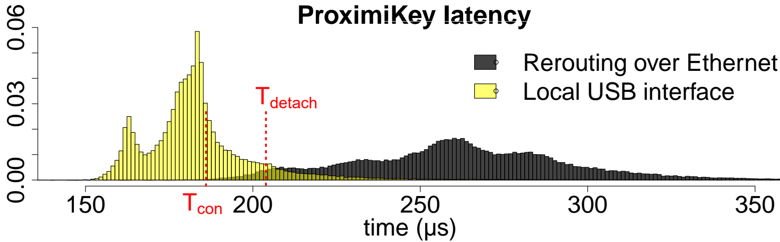


Figure 6.7: *Latency distributions: for legitimate challenge-response rounds (left) and simulated relay attack (right).*

samples are in between 209μs and 650 μs). Hence, the average delay of our simulated adversary is only 80μs. To put this into perspective, even the highly-optimized network connections between major data centers in the same region exhibit latencies from one millisecond upwards [125], which is one order of magnitude more than in our simulated setup.

Besides the latency observed on the side of the embedded device, we measured the time required to compute responses to received challenges on the side of the target platform. We repeated these tests on three different SGX platforms and observed results that varied from 6 to 10 μs. We also measured if the computational load of the target platform influences the time required to compute responses. Under maximum system load (all 8 cores busy), the maximum observed time increased to 20 μs. Under moderate system load (1 or 2 cores busy), we experience no notable increase in the required computation time.

6.5.5 Initial Proximity Verification Parameters

As explained in Section 6.4.3, the initial proximity verification is successful when at least a fraction k of the n challenge-response latencies are below the threshold T_{con}. Now, we explain our strategy for setting these parameters based on the above results.

There are five interlinked parameters that one needs to consider: (i) the legitimate connection latency threshold T_{con}, (ii) the total number of the challenge-response rounds n, (iii) the fraction k, (iv) the attacker's success probability P_{adv} that should be negligible, and (v) the legitimate success probability P_{legit} that should be high. We find suitable values for these parameters in the following order:

1. We start with the threshold T_{con}. The higher T_{con} is, the higher the legitimate success probability P_{legit} becomes. On the other hand, a too-

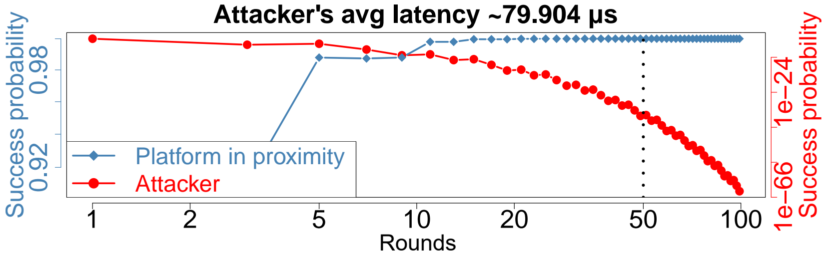


Figure 6.8: *Parameter tuning: the attacker's success probability P_{adv} and the legitimate success probability P_{legit} for different numbers of rounds n given a fixed k.*

high value for T_{con} also makes P_{adv}, the attacker's success probability, high. Therefore, we are after a suitable value for T_{con} that keeps P_{legit} high while minimizing P_{adv} over a varied number of rounds n.

2. Based on such T_{con}, we pick a fraction k such that it maximizes the legitimate success probability P_{legit} and reduces the attacker's success probability P_{adv}.

3. Given T_{con} and k, we evaluate P_{adv} and P_{legit} over a varied number of rounds n and choose the minimum number of rounds that provides the required probabilities since the fewer rounds, the faster the initial attestation is.

Main Result. Figure 6.8 shows the legitimate enclave's success probability P_{legit} and the attacker's success probability P_{adv} with different numbers of rounds. Based on our experiments, we set $T_{con} = 186\mu s$ (see Figure 6.7), the threshold fraction $k = 0.3$, and the number of rounds $n = 50$, which yields a very high legitimate success probability $P_{legit} = 0.999999977$ and a negligible attacker's success probability $P_{adv} = 3.55 \times 10^{-34}$.

6.5.6 Periodic Proximity Verification Parameters

For periodic proximity verification, we have two main requirements. First, the attacker's success probability P'_{adv} must be negligible. Recall that P'_{adv} refers to an event where the device is detached, but the connection is not terminated sufficiently fast. Second, the probability of false positives P'_{fp} should be very low. P'_{fp} refers to an event where the connection is terminated when the device is still attached. Next, we explain the three-

step process to set up parameters T_{detach}, w, and f for the periodic proximity verification:

1. We find out a suitable latency T_{detach} that defines the yellow or red round in Figure 6.5. The yellow window defines the round of challenge-response latency between T_{con} and T_{detach}, while the red window defines a latency more than T_{detach}. Hence, the probabilities $\Pr[T_{con} \leq \mathcal{L}_{legit} \leq T_{detach}] = \Pr[legit \in \text{yellow}]$, and $\Pr[\mathcal{L}_{legit} \geq T_{detach}] = \Pr[legit \in \text{red}]$ should be very low. $\mathcal{L}_{legit}$ and $\mathcal{L}_A$ denote the latency of the legitimate enclave running on the platform in proximity and the remote attacker platform's latency, respectively.

2. Based on the threshold T_{detach}, we select a suitable sliding window size w to minimize the attacker success probability P'_{adv} to a negligible quantity.

3. We fix a suitable frequency f for the periodic challenges. A high f value terminates the communication very fast, leaving very small attacking window.

Main Result. Based on the above strategy, we set the periodic proximity verification parameters as follows: $\Pr[A \in \text{success window}] = P'_{adv} = P'_{fn} = 3.55 \times 10^{-34}$, $\Pr[legit \in \text{success window}] = 0.999999977$ and $\Pr[legit \in \text{failed window}] = P'_{fp} = \Pr[legit \in \text{red}]^2 = 1.6 \times 10^{-4}$ and $T_{detach} = 205\mu s$ (see Figure 6.7). If at least two latencies above T_{detach} are received, the ProximiKey terminates the connection and revokes the platform. The average downtime due to false positives occurring during a connection of 10 years is around 2 minutes.

6.5.7 Performance Analysis

In addition, we evaluated the following two performance metrics:

1. *Start-up latency.* The initial proximity verification takes 2 ms. The complete connection establishment, including attestation and TLS handshake, takes less than 1 second.

2. *Operational latency and data overhead.* Our solution adds around $200\mu s$ of additional latency for TLS and transport over the native USB interface of the FX3. The data overhead is around 80 bytes per packet for the header and the MAC. Execution of the periodic ProximiTEE protocol with 83 rounds/second requires around 156.14 KBytes/s of data which is only 2.4×10^{-3}% of the USB 3.0 channel capacity.

6.5.8 Preventing Relay to Co-Located Platform

The primary purpose of our experimental evaluation was to show that our inexpensive PROXIMITEE prototype can effectively prevent relay attacks where the adversary redirects the attestation to another platform that is under their physical control in a *different location*. Next, we discuss whether PROXIMITEE can prevent attestation redirection to a *co-located* platform, like another server on the same server rack.

If the two co-located platforms are connected through traditional networking technologies like Ethernet (as in our experiments), our evaluation already shows that such relay attacks can be effectively prevented using a simple and inexpensive embedded device like our prototype. However, in some modern data centers, computing platforms are connected with faster interconnect technologies like InfiniBand connections that can enable latencies as low as $7\mu s$ [126].

The ability to distinguish relay attacks depends on three key factors. The first is the latency of the channel through which the relay is performed (e.g., $7\mu s$ for InfiniBand). The second is the time required to compute responses to challenges on the target platform (e.g., $6-10\mu s$ in the SGX platforms that we tested). And the third is how much variance the round-trip times between the embedded device and the target platform have (e.g., $10-20\mu s$ in our USB 3.0 prototype). The local communication variance and the response computation time should be less than the relay latency to enable robust proximity verification.

We conclude that our simple prototype cannot prevent all possible relays to co-located platforms when high-speed interconnect technologies such as InfiniBand are used. To address such relay attacks, one needs a faster and more accurate embedded device that exhibits less variance. For example, PCIe-connected FPGAs can have latencies as low as $1\mu s$ [127]. Besides a better embedded device, one can also increase the number of distance-bounding protocol rounds and reduce the success probability for legitimate attestation P_{legit}.

6.6 Addressing Emulation Attacks

We consider attestation key extraction from SGX processors difficult and rare, in contrast to the previously considered relay attacks that require only OS control or other malicious software on the target platform. However, attacks like Foreshadow [120] have shown that extracting attestation keys from SGX processors is not far-fetched. Although Intel has the possibility to issue microcode patches that address processor

vulnerabilities like Meltdown and the processor's microcode version is reflected in the SGX attestation signature, new vulnerabilities like the ZombieLoad attack [128] may be discovered. Before microcode patches are deployed, on some occasions, leaked but not revoked attestation keys may be available to the adversary.

6.6.1 Emulation Attack

Adversary Model. We consider an *emulation attacker* has all the capabilities of the relay attacker (cf. Section 6.3) and additionally has obtained at least one valid (not yet revoked by Intel) attestation key from any SGX platform but the target platform. The adversary might obtain an attestation key by attacking one of their processors or by purchasing an extracted key from another party.

The Emulation Attack. In the attack, the adversary uses a leaked attestation key to emulate an SGX processor on the target platform. Since the IAS (or any other attestation service) successfully attests the emulated enclave, it is impossible for the remote verifier to distinguish between the emulated enclave and the real one.

Emulation Attack Implications. The emulation attack allows the adversary to fully control the attested execution environment and thus break two of the fundamental security guarantees of SGX – enclave data confidentiality and code integrity – and to access any secrets provisioned to the emulated enclave. Since the OS is also under the control of the attacker, any attempted communication with the real enclave will always be redirected to the emulated enclave.

6.6.2 Boot-Time Initialization Solution

Proximity verification alone cannot protect against the emulation attacker, as the locally emulated enclave would pass the proximity test. Therefore, we describe a second hardened attestation mechanism that leverages secure boot-time initialization and is designed to prevent emulation attacks. This solution can be seen as a *novel variant* of the well-known TOFU principle, with the main benefit of our solution over previous variants being that it simplifies deployment and increases security. Additionally, when such attestation is used in combination with our previously described periodic proximity verification, our solution enables secure offline revocation.

Security Assumptions. Our security assumptions regarding the target platform are described in Section 6.3. The only difference is that, in this case, we assume that the UEFI (or BIOS) on the target platform is trusted.

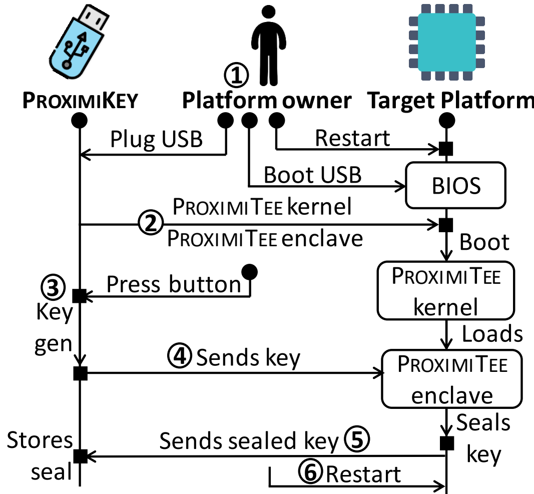


Figure 6.9: *Boot-time initialization: The* ProximiKey *uses a minimal kernel Linux image to boot and load the* ProximiTEE *enclave on the target platform and seal a platform-specific secret to the* ProximiKey *memory.*

Solution Overview. Figure 6.10 illustrates an overview of this solution. During initialization, which is depicted in Figure 6.9, the target platform is booted from the attached device that loads a minimal and single-purpose ProximiTEE kernel on the target device. In particular, this kernel includes no network functionality. The kernel starts the ProximiTEE enclave, which shares a secret with the device. This shared secret later bootstraps the secure communication between ProximiKey and the ProximiTEE enclave. *The security of the bootstrapping relies on the fact that the minimal kernel will not perform enclave emulation at boot time.* The ProximiTEE enclave will later be used as a proxy to attest whether other (application-specific) enclaves in the system are real or emulated and on the same platform.

Boot-Time Initialization. The boot-time initialization process is performed only once. This process is depicted in Figure 6.9, and it proceeds as follows:

① The platform owner plugs ProximiKey into the target platform, restarts it to BIOS, and selects the option to boot from ProximiKey.

② ProximiKey loads the ProximiTEE kernel and boots from it. The ProximiTEE kernel starts the ProximiTEE enclave.

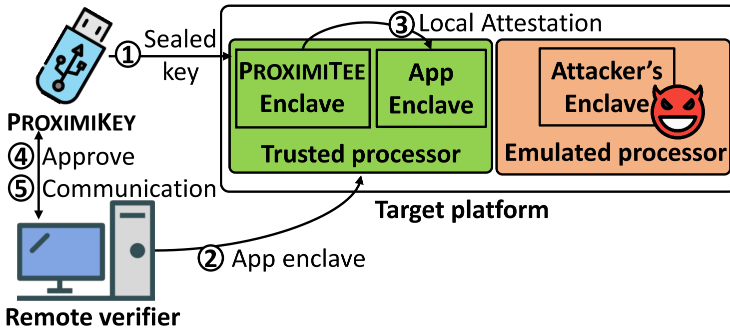


Figure 6.10: PROXIMITEE *boot-time attestation: After the boot-time initialization (refer to Figure 6.9), the* PROXIMITEE *enclave executes a local attestation with the verifier uploaded application-specific enclave.*

③ The user presses a button on PROXIMIKEY to confirm that this is a boot-initialization process. This step is necessary to prevent an attack where the compromised OS emulates a system boot.

④ PROXIMIKEY sends a randomly generated key $\mathcal{K}$ to the PROXIMITEE enclave.

⑤ The enclave returns the sealed key $\mathcal{S}$ corresponding to the key $\mathcal{K}$ ($\mathcal{S} \leftarrow \texttt{Seal}(\mathcal{K})$) to PROXIMIKEY, which then stores the key and the seal pair $(\mathcal{K}, \mathcal{S})$ on its flash storage.

⑥ PROXIMIKEY blocks further initializations, sends a restart signal, and boots the platform with the regular OS.

Attestation Process. After initialization, the target platform runs a regular OS. The attestation process is depicted in Figure 6.10 and proceeds as follows:

① PROXIMIKEY sends the seal $\mathcal{S}$ to the PROXIMITEE enclave that unseals it and retrieves the key $\mathcal{K}$. PROXIMIKEY and the PROXIMITEE enclave establish a secure channel (TLS) using $\mathcal{K}$.

② The remote verifier uploads a new application-specific enclave on the target platform.

③ The PROXIMITEE enclave performs local attestation (see Section 6.2) on the application-specific enclave that binds its public key to the attestation.

④ The ProximiTEE enclave sends the measurement and the public key of the application-specific enclave to ProximiKey. ProximiKey establishes a secure channel to the application-specific enclave and sends the measurement of the enclave to the remote verifier. The remote verifier then approves the communication to the application-specific enclave.

⑤ The remote verifier checks that the measurement of the application-specific enclave is as expected. If this is the case, it can communicate with the enclave through ProximiKey.

Following Communications. Similar to our previous solution, after the initial attestation, all the communication between a remote verifier and the enclave is mediated by the ProximiKey which periodically checks the proximity of the attested enclave and terminates the communication channel in case the embedded device is detached.

6.6.3 Security Analysis and Implementation

In this attestation mechanism, the task of establishing a secure communication channel to the correct enclave can be broken into three subtasks. The first subtask is to establish a secure channel to the correct ProximiKey device. In our solution, this is achieved using standard device certification. Recall that the adversary cannot compromise the specific ProximiKey used.

The second subtask is to establish a secure communication channel from ProximiKey to the ProximiTEE enclave. The ProximiKey shares a key with an enclave that is started by the trusted ProximiTEE kernel, hence at a time in which the attacker could not emulate any enclave. The ProximiKey knows when secure initialization takes place, as the platform owner indicates this by pressing a button – an operation that the adversary cannot perform. The ProximiTEE enclave seals the key during initialization. Different SGX CPUs cannot unseal each other's data, and therefore even if the adversary has extracted sealing keys from other SGX processors, they cannot unseal the key and masquerade as the legitimate ProximiTEE enclave.

The third subtask is to establish a secure communication channel from the ProximiTEE enclave to the application-specific enclave. The security of this step relies on SGX's built-in local attestation. An adversary in possession of leaked sealing attestation keys from other SGX processors cannot produce a local attestation report that the ProximiTEE enclave would accept, and

therefore the adversary cannot trick the remote verifier into establishing a secure communication channel to a wrong enclave.

Comparison to TOFU. Our second attestation mechanism is a novel variant of the well-known "trust on first use" principle. In this section, we briefly explain the main benefits of our solution over common TOFU variants.

1. **Smaller TCB size and attack surface.** In the TOFU solution, the standard and general-purpose OS needs to be trusted on first use, and the CA needs to remain online for enrollment of new SGX platforms. In our solution, a significantly smaller and single-purpose kernel needs to be trusted on first use. Additionally, we require trust in the BIOS (or UEFI). In our solution, the CA can remain offline when a new platform is enrolled.

2. **Reboot instead of re-install.** Our solution requires that the target platform is rebooted once from PROXIMIKEY. In most TOFU solutions, the target platform requires a clean state which is difficult to achieve without re-install, which makes deployment difficult.

3. **Secure offline revocation.** When boot-time initialization is combined with the previously explained periodic proximity verification, our solution provides an additional property of secure offline revocation that requires no interaction with the CA. Such property is missing from previous TOFU solutions.

Implementation. We implemented a complete prototype of our second attestation mechanism. On top of our previous PROXIMITEE implementation (see Section 6.5.1), the boot-time initialization solution requires the PROXIMITEE kernel. We have modified an image of Tiny Core Linux [129] and used it as the boot image for our boot-time initialization. The image size of our modified Linux distribution is 14 MB (in contrast to 2 GB standard 64-bit Linux images build on the standard kernel). Our image supports bare minimum functionality and includes `libusb`, `gcc`, Intel SGX SDK, Intel SGX platform software (PSW), and Intel SGX Linux driver. The PROXIMITEE enclave is a minimal enclave that uses a simple serial library to communicate with the PROXIMIKEY and a local attestation mechanism to attest any application-specific enclave.

6.7 Discussion and Related Work

Extension to other TEEs. Our approach could be applied to other TEEs as well. The critical requirement for the TEE is that it must support

programmable operations that can be executed sufficiently fast. One TEE that meets these requirements is ARM TrustZone.

DRTM Proximity Verification. Presence attestation [130] enables proximity verification of DRTM-based TEEs [6]. The TEE shows an image that is captured by a trusted camera and communicated to a remote verifier. The same approach cannot be used with SGX since it lacks a trusted path for secure image output. Catching the cuckoo [131] uses timing side channels to verify proximity with a TPM emulator. The TPM latency is in the order of seconds, making it infeasible for any practical distance bounding.

6.8 Conclusions

Relay attacks have been known for more than a decade, but their implications for modern TEEs like SGX have not been carefully analyzed. In this chapter, we have presented the first such analysis and shown that attestation redirection increases the adversary's ability to attack an attested enclave. We have also proposed ProximiTEE as a solution to prevent relay attacks using proximity verification. Our experimental evaluation is the first to show that proximity verification can be made secure and reliable for TEEs like SGX. As an additional contribution, we have also presented a novel boot-time initialization solution for addressing a stronger emulation attacker who has leaked attestation keys.

Chapter 7
Runtime Trust in Intel SGX

7.1 Introduction

While generally, one needs to trust the manufacturers with the implementation of the CPU and, therefore, of the TEEs (e.g., SGX), trust assumptions in the context of TEEs are more nuanced. In this chapter, we distinguish between manufacture-time and deployment (runtime) trust assumptions and further analyze those assumptions in the context of TEE-based isolation, attestation, and sealing.

This distinction is maybe best illustrated through examples of sealing and remote attestation. Sealing is typically supported by keys generated at manufacture time that are stored only locally within a CPU. Sealing keys, therefore, cannot be exported out of the CPU (not even to the manufacturer), guaranteeing that the manufacturer cannot access the confidential data that the user sealed. Thus, for sealing, the users need to trust the manufacturer only at manufacture-time but not after deployment (i.e., at runtime). This has real-world implications. If sealing is implemented as described above, a manufacturer would not be able to provide access to users' confidential data, not even under subpoena.

Remote attestation, however, typically assumes trust in the manufacturer at both manufacture time and runtime. This is due to the manufacturers preserving the ability to revoke and certify CPUs after they have been deployed.

A manufacturer that is honest at manufacture-time can therefore aim to implement isolation, local attestation, and sealing such that these are only vulnerable at manufacture time but not vulnerable at runtime. Intel SGX follows the trust model described above and claims not to have access to the CPU (root) sealing key [132, 133, 45].

In this chapter, we challenge this claim, and we describe an attack that allows Intel to extract the CPU Root Sealing Key (RSK) as well as the Owner Epoch Key (OEK) from any SGX CPU and, therefore, to unseal all data that enclaves sealed on the victim CPU. In addition, this attack allows Intel to brake local attestation and to compromise remote attestation with more stealth than so far assumed.

7.2 Attacker Model: Manufacture-Time vs. Post-Deployment Compromise

We assume that Intel is not malicious or compromised during manufacturing. What we mean by this is that the process as described in [45] is followed, and the CPU is provisioned with two secret keys in its e-FUSES: a root provisioning key (RPK) and a root seal key (RSK). The RPK is retained by Intel, while the RSK is not. However, we consider an attacker who, after CPU deployment, obtains access to the RPK. This access can be the result of a compromise or collusion with Intel or a consequence of Intel being compelled to provide access to the RPK (e.g., by a court order). In summary, we assume that the RPK is not compromised at manufacture time but is compromised post-deployment.

Finally, we assume that the attacker is able to run an enclave in the victim CPU, which can obtain all key types from `EGETKEY` (cf. Section 7.3.1), even if only in debug mode. As we explain below, on older CPUs without support for *SGX Flexible Launch Control* (FCL), the attacker can get all key types if they have at least a debug [1] launch enclave *signed by Intel*. On newer CPUs with support for SGX FCL, the attacker just needs supervisor privileges on the target system.

The goal of the attacker is to recover secrets that have been sealed by client enclaves before the compromise of the RPK and to break local and remote attestation on the target CPU.

Note that the attacks we describe next are based on publicly available information, such as patents and manuals. Real implementation might differ from the documents we analyzed, but nonetheless, our analysis still serves as a warning of what can concretely go amiss in such deployments.

7.3 Background

7.3.1 Intel SGX EGETKEY

`EGETKEY` is one of the SGX `ENCLU` instruction's leaf functions, and it is used to generate cryptographic keys. As is publicly documented in Chapter 36 of Volume 3D of [37], `EGETKEY` can be used to generate five key types, which serve different purposes. These key types are used to seal secrets to untrusted storage (`PROVISION_SEAL_KEY`, `SEAL_KEY`), control enclave

[1] The enclave can be the default one present on github `https://github.com/intel/linux-sgx/tree/master/psw/ae/le`, but with an `ATTRIBUTE_MASK`, which allows for debugging. A debug provisioning enclave works as well, but then the attacker needs supervisor privileges as well.

launch policies (EINITTOKEN_KEY), perform local attestation (REPORT_KEY), and as a shared secret with Intel (PROVISION_KEY). EGETKEY enforces several restrictions on which keys can be obtained and in which context. Particularly, EGETKEY can only be executed by an SGX enclave (also when in debug mode), and it only provides the SEAL_KEY and the REPORT_KEY unless the calling enclave is provisioned with special attributes. For instance, EGETKEY returns the PROVISION_KEY and PROVISION_SEAL_KEY if and only if the calling enclave context has the attribute ATTRIBUTES.PROVISIONKEY set to 1. An enclave with this attributes set can be started only if at least one of the following two conditions is met: i) it is signed by the same signer stored in the IA32_SGXLEPUBKEYHASH MSR, ii) it is approved by an already running launch enclave. On systems with Intel FCL enabled, this implies that launching a (debug) enclave with ATTRIBUTES.PROVISIONKEY set can only be done if the attacker has supervisor privileges. On systems without FCL, the attacker either needs to have an arbitrary enclave signed by Intel with ATTRIBUTES.PROVISIONKEY set, or one of the default launch enclaves or provisioning enclaves with debugging enabled. With debugging enabled, the adversary can then read the generated debug keys from the enclave memory or let the debug launch enclave approve a custom debug enclave with ATTRIBUTES.PROVISIONKEY set.

EGETKEY Algorithm. According to Intel's patents [132, 133], EGETKEY follows the recommendations of NIST SP 800-108 [134] for its key derivation function (KDF). In constructing the KDF, NIST SP 800-108 requires a pseudo-random function (PRF). According to the patents [132, 133], the PRF chosen for EGETKEY is AES-CMAC [135]. The patents also refer to a *parent key* used to key AES-CMAC. This *parent key*, according to the EPID attestation whitepaper [45], is derived from the Root Provisioning Key (RPK) and the current CPU security version number (CPUSVN), which tracks the security version of the CPU logic (e.g., microcode, and XuCode [44]). The RPK is one of two keys that are present on the CPU fuses [45], the second one being the root seal key (RSK). Each CPU has its unique RPK and RSK, and according to public documents [132, 133, 45], Intel retains the RPK but does not know the RSK. Since by knowing the RPK an attacker can derive the CPUSVN-specific key, unless otherwise specified, for the rest of this chapter, we refer to the RPK as the key used to key AES-CMAC (and not the actual key stored in the e-FUSES).

In short, EGETKEY runs AES-CMAC to derive keys. On each CPU, the AES-CMAC is keyed with that CPU's RPK, which is known to Intel (for every

CPUSVN). Keys derived by `EGETKEY` are essentially the MAC tag obtained over a message given to the AES-CMAC keyed with the RPK.

EGETKEY Parameters. Different key types, as introduced above, are obtained by changing the *fields* included in the message given as input to the `EGETKEY` AES-CMAC. Message *fields* can be, for example, an enclave measurement, attributes, developer ID, and the CPU RSK, to name a few. If a key type does not use a particular field, that field is replaced with a zero bitstring of the same length. Note that two enclaves that ask `EGETKEY` for the same key type do not, in general, get the same key. This is because even if the fields used are the same, the fields' content would, in general, be different. For instance, even when asking for the same key type, a debug enclave does not get the same key as its corresponding non-debug enclave because the debug attribute field (which is part of the AES-CMAC input message) is set to 1 (0) in the (non-)debug enclave.

Almost all of the message fields are known to an `EGETKEY` caller. There are only two fields that are secret: the RSK, which is unknown to everyone but the CPU; and the `OWNER_EPOCH` key (OEK), which is known only to the platform owner. Therefore, an attacker that knows only the RPK still cannot generate all of the keys that depend either on the RSK or the OEK. The only key that does not depend on the RSK or the OEK is the `PROVISION_KEY`.

7.3.2 CMAC

In the following, we assume a standard implementation of CMAC as described in [135]. We summarize the general notation used throughout this chapter in Table 7.1. As we will need some AES-CMAC details in the following sections, below we give the equation used to compute C_i from [135] and depict in Figure 7.1 the main steps of the CMAC algorithm.

$$C_i = AES_K(C_{i-1} \oplus Mi) \tag{7.1}$$

7.3.3 Recovering a Message Block from AES-CMAC

In this section, we consider an unusual attacker model for AES-CMAC. Usually, the *verifier* is trusted, and its role in the protocol is to verify whether a message tag is correct by knowing *the key and the message*. However, as this is relevant within the `EGETKEY` context, we instead consider a scenario in which an AES-CMAC *verifier*, with access to the CMAC key, can recover a message block M_i by only knowing part of the message and its tag. Formally, let us consider the scenario in which the verifier knows a MAC tag but, say,

Table 7.1: *Summary of notation*

Symbols & Operators	Description
K	The secret key used in the CMAC cipher
$K1$	The first subkey derived from K in CMAC
$K2$	The second subkey derived from K in CMAC
M	Input Message to CMAC
$Mlen$	Length of the message M
$Tlen$	Output length of the CMAC
b	Block size of the CMAC cipher
n	Number of message blocks: $n \doteq \lceil Mlen/b \rceil$
0^s	A zero bitstring of length s bits
$X\|\|Y$	Bitstring resulting from the concatenation of bitstrings X and Y
$X \oplus Y$	Bitstring resulting from the bitwise exclusive OR of bitstrings X and Y
M_i	The i-th block of M, where each block is of length b bits
M_n^*	The last block of M, which is a partial block if $Mlen$ is not a multiple of b
M_n	The last block used in the CMAC computation. If M_n^* is a complete block, $M_n \doteq K1 \oplus M_n^*$; else, $M_n \doteq K2 \oplus (M_n^* \|\| 1 \|\| 0^{(nb-Mlen-1)})$
C_i	Intermediate value of the CMAC computation after processing message block M_i as defined in equation (7.1).
$\text{AES}_K(X)$	AES encryption of bitstring X with key K
$\text{AES}_K^{-1}(X)$	AES decryption of bitstring X with key K
RPK	Root Provisioning Key
RSK	Root Seal Key
OEK	Owner Epoch Key

does not know message block M_i of the message from which the tag was computed. In other words, the verifier does not know M_i but knows *all* M_j for $\forall j \neq i$ and $i, j \in \{k \in \mathbb{N} : 0 < k \leq n\}$.

Note that the verifier is supposed to know the key K in order to verify the integrity of the message. In the following, we refer interchangeably to the verifier as verifier, attacker, or adversary, and we will assume that $Tlen$ is equal to b. For instance, this means that if AES-128 is used, then $Tlen$, the length of the MAC tag, is also 128 bits (see Section 7.3.2 for the definition of the notation used in this section).

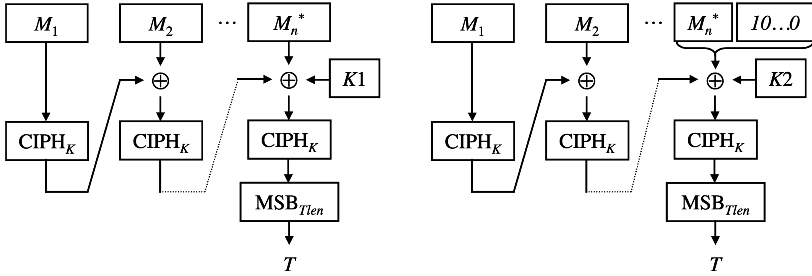


Figure 7.1: *Figure from [135] depicting the two cases for CMAC computation. The case on the left occurs if the message length is a multiple of the block size; otherwise, the case on the right is used.*

We will start by isolating M_i from equation (7.1), which results in the following equation:

$$M_i = AES_K^{-1}(C_i) \oplus C_{i-1} \tag{7.2}$$

where with AES_K^{-1} we denote the AES decryption with key K. Note that this is possible because AES can be inverted, that is, $AES_K^{-1}(AES_K(x)) = x$. As can be seen in equation (7.2), M_i depends on C_i and C_{i-1}. To start with, it should be noted that the verifier can trivially compute C_{i-1} by performing AES-CMAC with equation (7.1) up until index $i-1$. This is because this part of the computation only depends on K and the message blocks $M_{j<i}$, all of which are known by the attacker in our scenario. However, the adversary cannot continue with the normal AES-CMAC computation since to compute C_i from equation (7.1), besides C_{i-1}, they would need to know M_i, which is the message block they are trying to recover.

To compute C_i, we can isolate for C_{i-1} in equation (7.1). This results in $C_{i-1} = AES_K^{-1}(C_i) \oplus M_i$, from which we obtain the following equation by incrementing the index by one:

$$C_i = AES_K^{-1}(C_{i+1}) \oplus M_{i+1} \tag{7.3}$$

Essentially, this equation allows us to compute AES-CMAC *backwards*. Note that C_i in this equation depends on M_{i+1}, which is known by the attacker, and on C_{i+1}, which by recursion depends on all $M_{j>i+1}$ and on C_n which is the MAC tag (since $Tlen = b$). Therefore, iterating from the MAC tag backwards, the attacker can use equation (7.3) to compute C_i and then plug it into equation (7.2) together with C_{i-1} to obtain M_i, the block that was unknown at the beginning.

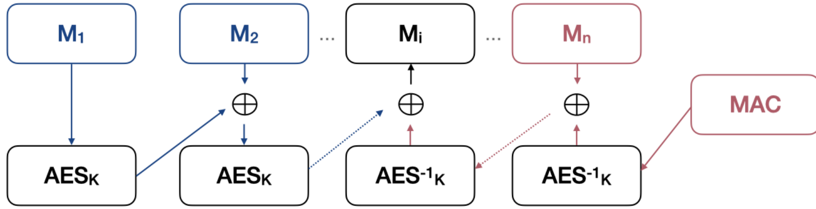


Figure 7.2: *Representation of the steps used to recover a block message M_i from an AES-CMAC tag. The blue arrows are computed from the first message block towards the middle, iterating over the blue message blocks with equation* (7.1)*. The red arrows are computed from the end (the MAC tag) towards the middle, iterating over the red message blocks with equation* (7.3)*. By combining these two steps, we obtain M_i.*

Figure 7.2 visualizes the computation performed to recover M_i. Overall, this attack only requires the adversary to obtain one AES-CMAC MAC tag for the message for which they want to recover the missing block.

So far, we assumed that the unknown bits to be recovered were perfectly aligned in a message block M_i. In Section 7.4, we discuss how the attack generalizes when this is not the case.

7.4 Special Cases for Message Recovery

In this section, we discuss how the AES-CMAC block recovery described in Section 7.3.3 can be generalized to the case in which the unknown b bits are not perfectly aligned with an AES-CMAC message block.

Formally, let us assume that the attacker does not know both the last m bits of M_{i-1} and the first $b-m$ bits of M_i, for $0 \leq m < b$. Using equation (7.1), the attacker can only compute up to C_{i-2}, while using equation (7.3) they can only compute until C_{i+1}. The attacker now needs to brute force the missing bits. However, they just need to brute force $2^{(min(m,b-m))}$ bits instead of 2^b bits. Note that if AES-128 is used, in the worst case, this reduces the number of tries from 2^{128} to 2^{64}.

To explain why, let us assume, without loss of generality, that $m < b-m$. Now the attacker can compute all the possible M_{i-1}, and therefore all the possible C_{i-1}, just with 2^m tries. After C_{i-1} is known, M_i can be derived with the method described in Section 7.3.3. However, among the resulting M_i, not all of the bits known to the attacker will match, therefore revealing that the guess made for the m bits of M_{i-1} was wrong. Note that there could be multiple guesses for which the known bits of M_i match. To filter out

those cases, we can use a second MAC tag computed over a message that only differs from the first one on at least one known bit. Now the adversary can use the previous valid guesses to compute the AES-CMAC tag over the second message. If the adversary computed tag matches the actual tag, the adversary knows that the guessed bits are correct.

7.5 Meet-in-the-Middle Attack on EGETKEY

7.5.1 Extracting the Root Seal Key

To extract the RSK, the attacker runs an enclave on the victim CPU (in production or in debug mode), which calls `EGETKEY` and outputs the `PROVISION_SEAL_KEY` to the attacker. We choose to call `EGETKEY` such to output `PROVISION_SEAL_KEY` since, in this call, the attacker knows all the inputs to `EGETKEY` except the RSK. Namely, `EGETKEY` calculates an AES-CMAC over its input (which also includes RSK) and outputs the `PROVISION_SEAL_KEY` as the AES-CMAC tag. The key that keys the AES-CMAC is derived from the Root Provisioning Key (RPK), which the attacker also knows.

In other words, the attacker knows the AES-CMAC key, the AES-CMAC tag, as well as the entire MAC input *except* for the 128 bits of the RSK. Hence, all the prerequisites to recover RSK from the AES-CMAC message, as described in Section 7.3.3, are satisfied. An attacker with access to the RPK (such as Intel) can therefore recover the RSK from any `PROVISION_SEAL_KEY` (even a debug one) and unseal any data from the target CPU.

7.5.2 Extracting the Owner Epoch Key

A similar procedure as described in the previous section can be followed to recover the OEK. However, as opposed to the RSK, there is no call to `EGETKEY` in which the only unknown message block to the attacker is the OEK. All the `EGETKEY` calls that take as input the OEK also include the RSK. The attacker can therefore extract the OEK only if they have already recovered the RSK.

Namely, to extract the OEK, the attacker can run an enclave that outputs two keys: the `PROVISION_SEAL_KEY` and then the `SEAL_KEY` [2]. The attacker first extracts the RSK from the `PROVISION_SEAL_KEY` and then, with the knowledge of RSK, extracts the OEK from the `SEAL_KEY`, following the steps from Section 7.3.3.

[2] The `REPORT_KEY` and the `EINITTOKEN_KEY` work just the same, as they also both depend on the RSK and OEK.

7.6 Implications of the Attack

In this section, we discuss the implications of an attacker leaking the RSK and/or the OEK on a target CPU. Note that by attacker, we refer to an attacker according to our attacker model (cf. Section 7.2), which among other things, knows the RPK and can execute code in the target system.

7.6.1 Breaking Sealing

Enclave sealing keys depend on both the RSK and the OEK of the CPU on which the enclaves run. If the attacker has leaked both RSK and OEK, it can unseal the secrets of all enclaves that ran on the victim CPU. This breaks the confidentiality of secrets stored by enclaves that might have run before the attack was even executed. The `SEAL_KEY` can be configured with several different fields, and thus, realistically, the attacker would also need to know the source code of the enclave for which they are trying to unseal secrets (if those parameters were not stored alongside the sealed blob). Note that if the enclave code is stored in the victim system, or if the target is well known, this should not be too challenging for the attacker. In general, any secret previously sealed on the CPU for which the attacker has leaked the RSK and OEK should be considered compromised.

7.6.2 Breaking Local Attestation

Local attestation relies on the `REPORT_KEY`, which depends on both the RSK and the OEK. An attacker that knows the latter two keys is able to generate valid MAC tags for arbitrary local attestation reports. Thus, this attacker is able to pass local attestation despite their code not running in SGX or even on the same CPU. The capability to fake local attestation completely breaks any SGX guarantee. For example, the attacker can fully emulate an enclave and easily pretend to be executing with SGX protections to other legitimate local enclaves. As a consequence, this allows to also fake remote attestation, as remote attestation relies on local attestation between the enclave to be attested and the quoting enclave. Note that this attack is **permanent**: the RSK cannot be updated (although the OEK can), so simply updating the microcode does not prevent this attack. The last statement further implies that even if a microcode update could prevent the attacks of Section 7.5, the attacker could still first downgrade to an older microcode and then run on the newer one while still retaining the correct RSK and OEK.

7.6.3 Breaking Remote Attestation

Knowledge of the RSK can be used to obtain the `PROVISION_SEAL_KEY`. The `PROVISION_SEAL_KEY` is used by the *Provisioning Enclave* and the *Quoting*

Enclave to, respectively, seal and unseal the current private EPID key. Hence, by compromising the RSK, the attacker can gain knowledge of the current EPID private key on the victim CPU. This allows the attacker to fake remote attestation, thus completely compromising new enclaves that are executed on the target system. This is because the attacker would be able to emulate the enclaves and then provide a legitimate remote attestation quote to the remote verifier, albeit the enclaves attested are actually not executing within an SGX environment.

Note that if the CPUSVN of the system is updated, the old EPID private key becomes invalid. To make the attack persistent, the attacker needs to be present on the victim system after a system update to unseal the new private EPID key.

Although it might not be surprising that Intel can manipulate remote attestation by, e.g., falsely attesting an enclave/CPU, the above attack is much more stealthy and applies both to unlinkable and linkable EPID attestations.

Chapter 8
Runtime Trust in AMD SEV-SNP

8.1 Introduction

Just like in Intel SGX, the attestation mechanism used by AMD also implicitly requires runtime trust in AMD. In the following, we describe how AMD can abuse this trust to compromise a deployed CPU TEE. For this analysis, we assume the same attacker model described in Section 7.2.

8.2 Background

AMD SEV-SNP relies on a co-processor, known as the PSP, to enforce its properties. Among its responsibilities, the PSP sets up and manages VM memory encryption and signs attestation reports. The VMs interact with the PSP through the hypervisor, which acts as an untrusted channel in the communication between VMs and PSP.

When the memory pages of a VM have been set up by the hypervisor, the hypervisor calls the `SNP_LAUNCH_UPDATE` function of the PSP. This finalizes the memory of the VM, and from that point on, the VM memory is protected from the hypervisor and other VMs by the encryption mechanism managed by the PSP. As part of `SNP_LAUNCH_UPDATE`, a special page of the VM is initialized by the PSP containing key material that can be used by the VM to establish a secure channel with the PSP.

Two keys are relevant for the runtime security of a SEV VM: the `VCEK` key and the `VEK` key. The `VEK` is generated at random by the PSP for each VM when the VM is started up and is used for VM memory encryption. The `VCEK` is instead used to sign the attestation report of the VM. The `VCEK` is a platform key – it does not depend on any VM measurement. The `VCEK` depends on secret values fused on the chip and on the `TCB_VERSION` number. The values that influence the `VCEK` are reported in Table 8.1.

8.3 Attacks and Implications

AMD can pursue (at least) two avenues to compromise a system at runtime, both of which are based on compromising the `VCEK`. If the `VCEK` is compromised, a VM can be launched without memory encryption while at the same time making a remote verifier believe that such memory encryption is in place. Without memory encryption, a malicious hypervisor can trivially leak the contents of the VM (and even modify them).

Table 8.1: *Values that influence the VCEK key. Taken from [47, p. 18]*

Bits	Field	Description
63 : 56	MICROCODE	Lowest current patch level of all cores
55 : 48	SNP	Version of the SNP firmware Security Version Number (SVN) of SNP firmware
47 : 16	-	Reserved
15 : 8	TEE	Current PSP OS version SVN of PSP operating system
7 : 0	BOOT_LOADER	Current bootloader version SVN of PSP bootloader

The first avenue relies on the fact that the TCB_VERSION number only contains the SVN of the PSP firmware and not an actual measurement of it. AMD can release a PSP firmware that launches SEV VMs in debug mode while at the same time not reporting this flag as set to the user. Another alternative would be to add a hidden function in the PSP that reveals a VM's VEK to the hypervisor. Note that since the PSP firmware version number is the same as the currently installed PSP firmware in the platform, the same VCEK is generated both with the benign and compromised PSP firmware. Thus there is no way for a verifier to distinguish these two cases based on attestation alone.

The second avenue relies on the current mechanism used to verify the authenticity of the VCEK. Particularly, in the current mechanism, the user needs to contact the AMD KDS server giving as argument the platform ID and the TCB_VERSION desired. The AMD KDS then replies with a certificate chain from an AMD root key to the public key of the VCEK associated with the given platform and TCB_VERSION. As AMD is able to generate every public key for a given platform, this avenue relies on the assumption that the related VCEK private key is also generated as part of the process. The internal mechanisms of how this process is implemented are not published, so it could be that the public key is generated without having access to the private key (e.g., an HSM that only outputs the public key could be used). However, if AMD can generate access to the private VCEK key, they can also trivially fake attestation if they become malicious at runtime.

Note that already running VMs can also be compromised even if they were instantiated by an uncompromised PSP firmware. This can happen if the VMs are deployed with VM migration enabled (something that is likely common in a cloud deployment). Then a VM migration can be started by the cloud provider to migrate the VM from a platform with a benign PSP

firmware to a platform with a compromised PSP or for which the VCEK is known. During the migration then either the VEK would be revealed by the malicious firmware, or the VM can be migrated to an emulated SEV environment that still passes attestation (given the adversary's knowledge of the VCEK in the new platform).

Chapter 9
Closing remarks

In this chapter, we summarize the main results of the thesis and discuss how future work can build on them to design more secure trusted execution environments (TEEs).

9.1 Conclusions

Allowing privileged software to manage some aspects of TEE execution is an appealing design decision. In principle, it allows the reduction of the software that has to be included in an enclave's TCB. For example, if the OS is in charge of memory paging, a complex software task, the enclave does not need to include any software to manage it itself. This separation also minimizes the changes that developers have to perform on their codebases to adapt to the new technology – the interface with the system remains very similar, and thus the integration is almost effortless. However, both previous work and the results of this thesis show that we lack an understanding of how a privileged attacker can abuse its capabilities to compromise TEEs. In this thesis, we contributed to this understanding in four main directions. In terms of data confidentiality, we showed in Chapter 3 that frequently issuing interrupts can lead to the CPU operating in an often understudied regime. In this regime, where the pipeline is frequently flushed, instructions exhibit timing characteristics that were not seen before and that could not be leveraged by unprivileged attackers. We showed how these timing characteristics lead to side channels that can extract cryptographic keys even from (supposedly) side-channel resistant libraries.

TEEs also aim at providing code confidentiality. Code confidentiality is often overlooked, but it is an important feature enabled by TEEs, as it is challenging to achieve it with other techniques. In Chapter 4, we developed a methodology to study to what extent they actually provide these guarantees in both Intel and AMD architectures. Perhaps surprisingly, we concluded that interpreted code is generally leakier in TEEs compared to executing instructions natively. We also note how this leakage is possible thanks to the capabilities afforded to a privileged attacker and would be considerably less otherwise.

As both of these previous attacks leverage the ability of the attacker to frequently send interrupts, we explored in Chapter 5 to what extent

AEX-Notify, a new Intel architecture extension, hampers these attacks. The proposal does not fundamentally reduce the capabilities of the attacker but only makes the enclave aware of a potential attack. While this is a step in the right direction, our analysis shows that it remains to be seen (based on actual implementations) to what degree this actually prevents interrupt-based attacks against Intel SGX.

In terms of how a privileged attacker can affect attestation, we analyze in Chapter 6 the impact that relay attacks have on the capabilities of the attacker. We show that relay attacks have the potential to enable physical attacks even if the attacker does not otherwise have access to the target platform. Particularly, the attacker can spend many resources physically compromising the die of one CPU and compromise deployments by redirecting execution on that CPU. Note that such expensive attacks are generally limited to one CPU, but relay attacks allow to amortize the cost by always allowing to redirect execution to the compromised CPU. To mitigate these attacks, we proposed ProximiTEE, which leverages proximity verification with a device connected to the target platform to prevent attacks.

Finally, in Chapter 7, we showed that current execution protocols implicitly assume runtime trust in the manufacturer, even if they often claim otherwise. We show how Intel can abuse its attestation protocol at runtime in Chapter 7 and how similar considerations apply to AMD SEV as well in Chapter 8. Particularly, these attacks work on the assumption that the manufacturers are in a more privileged position to extract CPU secrets that even they should not have access to. These attacks show that attestation protocols should be designed to take these threats into account.

9.2 Future work

In the following, we investigate three main directions that we believe should be explored to chase more secure TEE systems.

9.2.1 Exploring Further Attacker Avenues

As shown in Chapters 3 and 4, a privileged attacker can leverage many system interfaces to compromise a TEE, such as controlling interrupts and page table entries. However, other low-level OS interfaces could be further explored. As an example, albeit not explicitly, the OS has indirect control over the data placement in the cache and when and whether the cache gets flushed. Physical attackers [136] can further install cache-coherent devices in the system, which could further enhance the resolution of cache attacks

to levels not seen before. These attacks could even leverage the scale of data centers and their tendency to migrate VMs or expose memory between different machines. Our understanding of how the hypervisor and OS can impact enclaves in large-scale deployments is still quite limited, and, as this thesis has shown, it could be abused by the attacker in unexpected ways.

9.2.2 Decreasing the Capabilities of Privileged Attackers

As we discussed throughout the thesis, one of the main difficulties in protecting TEEs is due to the capabilities afforded to the attackers and their implicit control over the enclave environment. As discussed, this helps in reducing the TCB. Nonetheless, we believe that there is a potential for designs to offload some of the management tasks to the attacker while at the same time limiting its capabilities. For instance, designs where system performance is sacrificed in favor of a more restrictive TEE could be explored. These designs could remove or delay system-level capabilities until the enclave is done executing in at least some of the CPU cores. Further work is necessary to evaluate whether these proposals are effective or even worth paying the performance price.

9.2.3 Susceptibility to Side Channels

In current designs, as discussed in Chapters 3 and 4, the responsibility to protect against side channels is left to the developer. However, carefully accounting for side channels in software is deeply linked to the concrete hardware implementation. Albeit constant time code proposals help alleviate the problem, as discussed in Section 3.8, they are often impractical. Future TEE architecture could account for side channels in hardware and provide a more convenient ISA to developers to avoid them, if not in all execution contexts, at least inside the enclaves.

9.2.4 Attestation Enhancements

In Chapters 6 to 7, we discussed how current attestation protocols are unable to provide guarantees with respect to relay attacks and still require runtime trust in the manufacturer. While we proposed an enhancement to the attestation with ProximiTEE in Chapter 6, we believe that future attestation protocols should ship with relay prevention. More importantly, future protocols should do away with having to trust the manufacturer at runtime. There are several challenges in designing such protocols, particularly how to guarantee the authenticity of the key upon TCB updates, but we believe that tackling these problems can only enhance the security of future TEE deployments.

Appendices

Appendix A
Frontal Attack

A.1 Responsible Disclosure

We notified the Intel PSIRT on February 21 2020, about the Frontal attack. We sent them an initial report of the Frontal attack and a proof of concept for the vulnerabilities we identified. They informed us on April 22 that their best practices [65] already suggest avoiding secret-dependent branching, and therefore our attack is considered out-of-scope for their SGX libraries. In particular, they stated that the balanced branches of the IPP Crypto library we attack in Section 3.5.1 are not used for secret-dependent operations in the SGX architectural enclaves and hence do not pose any security implication. The vulnerability shown in Listing 3.1 was reported to the mbedTLS team, which promptly fixed it. The vulnerability was also described in a 2017 paper [16] and was still unknown to the developers.

A.2 Data-Oblivious Execution

Resilience against side-channel attacks is often a desired security property when implementing software. This property is particularly important for libraries and applications that operate on secret and sensitive data on a system controlled by the attacker. Side-channel attacks exploit secret-dependent variations of the program execution. These variations are generally of two types: control-flow-dependent and data-dependent. Control-flow secret dependencies are present whenever the control flow of an application depends on confidential information. Data dependencies manifest when latency or resources utilized depend on the input data. For example, when memory accesses at different addresses are performed based on some secret. Countless attacks have exploited these types of dependencies in the past [2, 137, 96], targeting in particular cryptographic libraries, as extracting secret keys handled by these libraries breaks any security guarantee built on top of them. Data oblivious execution defends against side-channel attacks by removing the two dependencies mentioned above. This eliminates any variation in program execution that would be potentially observable by the attacker. There are two ways to obtain a data-oblivious executable – first, writing it directly in low-level assembly code; second, by performing an automatic transformation at compile time

from a higher-level language. Note that writing the code in a higher-level language in a data-oblivious way and then simply compiling it might reintroduce data or control flow dependencies at the binary level.

Several techniques for compiling and transforming code from an arbitrary high-level language to data-oblivious code have been proposed [77, 138, 139, 140, 64]. One of the most complete constant-time transformations for SGX is Raccoon [77]. It removes any control flow and most data dependencies by transforming secret-dependent branches into a decoy and a real path that contain similar instructions. At run time, both paths are executed, allowing only the real one to modify memory by carefully applying the conditional move instruction (`cmov`). Raccoon runs on SGX enclaves and uses SGX's memory protections to ensure confidentiality against an attacker that can otherwise read arbitrary locations of memory.

A.3 Measurement Details

We made several changes from stock SGX-Step [67], primarily aiming to reduce measurement noise as much as possible. In terms of functionality, we added the possibility to measure performance counters alongside instructions' timings. We identified four major sources of noise: the OS, variability in the APIC timer, unpredictability of shared resource state, and enclave creation offset noise. We discuss how we addressed each of these in the following paragraphs.

The OS is a source of noise as it needs to run the scheduler on each core to decide which tasks to execute. If the scheduler runs in between the start of a measurement and its end, the measurement will inevitably be longer. Moreover, running any OS function while we single-step can sometimes evict part of the enclave memory from the cache, thus forcing the enclave to fetch it again when it is resumed. This also happens when the scheduler executes on the sibling core. As recommended in the original SGX-Step framework, we run the code in its own isolated core to reduce this noise. However, this alone stops neither the scheduler nor the other cores from interrupting the isolated core. We observed that disabling watchdogs at boot and disabling the graphical user interface tends to reduce the noise produced by the OS, albeit it does not eliminate it completely.

In stock SGX-Step, the APIC timer is set in the `aep_cb_func`. The `aep_trampoline` then executes and resumes the enclave. Various conditions can create variability between the time at which the APIC timer is set and the time at which the enclave resumes. For instance, sometimes,

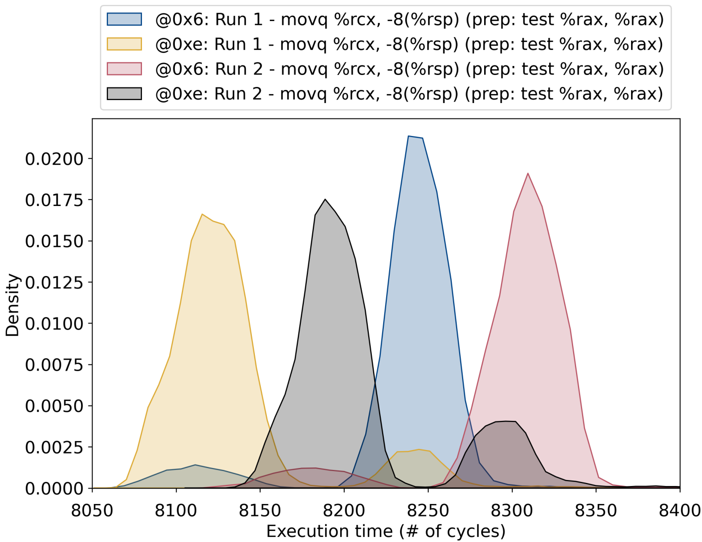


Figure A.1: *Distribution of the instructions of Figure 3.5 across different runs, split by their alignments. This figure highlights how different enclave runs exhibit a shift in the mean of their instructions' distribution, and hence distributions are not directly comparable between runs.*

the `aep_trampoline` code page or some of the data it uses might not be present in the cache. We addressed this variability by setting the APIC timer from the `aep_trampoline` function with a value passed from the `aep_cb_func` function and by serializing the instruction stream (using `CPUID`) just before setting the timer. Interestingly, while debugging for this source of noise, we observed that we were never able to interrupt in between fused macro-instructions as these seem to be treated atomically by the CPU, as also observed in [61].

The third source of noise stems from the difference in the microarchitectural state in-between measurements. While we could not completely eliminate this source of noise as we have no direct view of the microarchitecture, we were able to reduce it significantly. The most effective change in this regard was obtained by linearizing the code of the `aep_cb_func` so that there is no mis-speculation in between single steps and the function always has the same cache footprint. Even with this change, we observed that instructions that cross a virtual page boundary

```
1  loop_start:
2      mov     (%rcx, %rdx, 8), %rax
3      xor     %r9d, %r9d
4      cmp     %rsi, %rax
5      setb    %r9b
6      sub     %rsi, %rax
7      mov     %rax, (%rcx, %rdx, 8)
8      mov     (%rdi, %rdx, 8), %r8
9      mov     %r9, %rsi
10     cmp     %r8, %rax
11     adc     $0x0, %rsi
12     sub     %r8, %rax
13     mov     %rax, (%rcx, %rdx, 8)
14     add     $0x1, %rdx
15     cmp     %rdx, %rbp
16     jne     loop_start
```

Listing A.1: *Exploited `for` loop in the mbedTLS library's `mpi_montmul` function (compiled on `gcc 7.5.0` with `-O3`).*

remained noisy. To account for this, we remove these measurements from the trace when possible. Note that the attacker can easily tell if an instruction crosses the page boundary as the access bit of the new page is set by the CPU.

Finally, while validating these changes, we noticed a source of noise across enclave creations, whose effects we illustrate in Figure A.1. The figure shows the measurement of the `mov`s from Figure 3.5 across enclave creations. As can be seen, the distributions remain bimodal, but the position of the modes across creations changes. However, the relative position between the modes stays the same: the `mov` at alignment $0x6$ is slower than that at $0xe$ on both runs. While we never observed modes shifting more than 200 cycles, this shift is still large enough such that, for instance, the distribution of `nop` instructions could overlap with the distribution of multiplication instructions from different runs. Given this shifting between enclave creations, we concluded that instructions' timings *are only comparable within the same enclave*.

```
1  loop_start:
2      mov   (%rax), %rcx
3      sub   $0x8, %rax
4      mov   %rcx, %rdx
5      shl   $0x3f, %rcx
6      shr   $0x1, %rdx
7      or    %rdi, %rdx
8      mov   %rcx, %rdi
9      mov   %rdx, (%rax + 8)
10     cmp   %rax, %rsi
11     jnz   loop_start
```

Listing A.2: *Exploited `for` loop in the `mbedtls_rsa_gen_key` function of the mbedTLS library.*

A.4 Outside Intel SGX

A question remains on whether these effects manifest only while executing code inside an SGX enclave or whether they are present also while running a program outside of SGX. Since we cannot send interrupts fast enough during normal execution (outside SGX), we decided to simulate the effect of the interrupts by modifying the code in Listing 3.2 such that each `mov` triggers an exception. We handle the exception and measure the time it took to execute it, and then resume the program execution from the instruction after the one that triggered the exception. Note that exceptions are handled very similarly to interrupts, with the key difference that the instruction that is currently executing can retire when an interrupt is triggered, while it needs to be discarded when an exception is raised.

We observed that timing differences between instructions in the two branches were less pronounced than when the code was run within SGX. Nonetheless, we were able to observe a (small) correlation between the exception handling time of single instructions and the branch being executed. This correlation hints that the effects are not only present when interrupting enclaves but would also manifest when interrupting applications if we had a fast enough interrupt timer.

Appendix B
Code Leakage

B.1 Responsible Disclosure

On 2 November 2022, we disclosed our findings related to Chapter 4 to the following companies promising code confidentiality in TEEs: Veracruz (ARM) [23], Edgeless [24], Enarx [22], and Scone [25]. Edgeless acknowledged receiving our report but did not take any further steps. Enarx responded that they are researching mitigations, while Scone told us they are working on mitigating the reported issues. Veracruz responded that side channels are out of scope in their attacker model. Nonetheless, they are working on clarifying their documentation about the risks related to code confidentiality in TEEs.

B.2 x86 ISA Instruction Count

We focus only on the 64-bit version of the x86 architecture when creating candidate sets. In building the candidate sets for the microarchitectures supporting SGX and SEV, we need to account for the fact that some instructions are handled differently in these environments. Particularly, in SGX, some of the instructions are illegal and thus will never be called on bug-free enclaves. On SEV, all instructions are allowed to execute; however, they will cause a hypervisor intercept, thus leaking to the attacker which instruction was executed. In the case of SGX, we never include illegal instructions in a candidate set, while in the case of SEV, we place the intercepted instructions in candidate sets of size 1. Next, we detail what instructions exactly end up in this special classification for the two TEEs.

SGX. We used the information from the Intel SDM Manual [37] Volume 3D Table 35-1 to find the criteria for instructions not allowed in SGX. To summarize, instructions with a privilege level lower than 3 and instructions that perform I/O operations or that could access the segment register are considered Illegal. Note that an instruction could have an illegal version and a legal version. For instance, the `mov` instruction can write to the segment registers, and that version of the instruction is illegal.

SEV. Instructions that cause a hypervisor intercept on SEV are reported in "Table 15-7. Instruction Intercepts" of the AMD64 Architecture Programmer's Manual [38]. Note that there might be other conditions that cause intercepts, which might leak information to the attacker, but we only

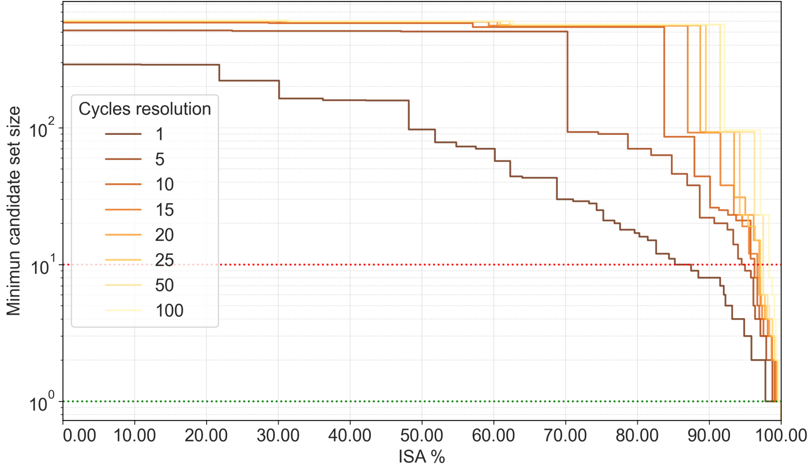


Figure B.1: *Candidate set sizes' distributions on the Skylake microarchitecture for a SotA attacker in the native system with varying cycle accuracy thresholds.*

consider the instructions on that table in our calculation. Finally, the dataset we used for the Zen microarchitecture was actually obtained from information collected from a Zen+ CPU from uops.info [107]. The Zen+ and Zen microarchitectures support the exact same x86 instructions; however, the Zen+ does not provide support for SEV.

B.3 Analysis of SotA Attacker Cycle Accuracy

To give an idea of the relationship between the strength of the SotA attacker's instruction cycle resolution and the native system information leakage, we show in Figure B.1 how the candidate set sizes change with different thresholds for the attacker's cycle resolution.

Bibliography

[1] J. B. Dennis and E. C. Van Horn. "Programming Semantics for Multiprogrammed Computations". *Commun. ACM* (1966).

[2] P. C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". *Advances in Cryptology — CRYPTO '96*.

[3] P. Gutmann. "Data Remanence in Semiconductor Devices". *10th USENIX Security Symposium (USENIX Security '01)*.

[4] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. "Architectural Support for Copy and Tamper Resistant Software". *SIGPLAN Not.* (2000).

[5] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing". *ACM International Conference on Supercomputing 25th Anniversary Volume*.

[6] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. "Flicker: An Execution Infrastructure for TCB Minimization". *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*.

[7] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. "How Low Can You Go? Recommendations for Hardware-Supported Minimal TCB Code Execution". *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*.

[8] ARM. *ARM Security Technology Building a Secure System using TrustZone Technology*. `https://developer.arm.com/documentation/PRD29-GENC-009492/c`.

[9] M. Schneider, R. J. Masti, S. Shinde, S. Capkun, and R. Perez. *SoK: Hardware-supported Trusted Execution Environments*. 2022.

[10] I. Corporation. *Intel Software Guard Extensions*. `https://software.intel.com/en-us/sgx`.

[11] Advanced Micro Devices Inc. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. `https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf`. Accessed: May 2022.

[12] ARM. *Arm Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A*. `https://developer.arm.com/documentation/ddi0615`.

[13] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys ’20)*.

[14] S. Checkoway and H. Shacham. “Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface”. *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’13)*.

[15] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. *11th USENIX Workshop on Offensive Technologies (WOOT ’17)*.

[16] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing”. *26th USENIX Security Symposium (USENIX Security ’17)*.

[17] Y. Xu, W. Cui, and M. Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. *2015 IEEE Symposium on Security and Privacy*.

[18] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. “Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX”. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS ’17)*.

[19] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. “Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. *26th USENIX Security Symposium (USENIX Security ’17)*.

[20] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. *2020 IEEE Symposium on Security and Privacy (SP)*.

[21] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia. “VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface”. *30th USENIX Security Symposium (USENIX Security 21)*.

[22] *Enarx: WebAssembly + Confidential Computing*. `https://enarx.dev`.

[23] M. Brossard, G. Bryant, B. El Gaabouri, X. Fan, A. Ferreira, E. Grimley-Evans, C. Haster, E. Johnson, D. Miller, F. Mo, D. P. Mulligan, N. Spinale, E. van Hensbergen, H. J. M. Vincent, and S. Xiong. *Private delegated computations using strong isolation*. Technical report. 2022.

[24] *Edgeless Systems: Confidential computing at scale for everyone*. `https://www.edgeless.systems`.

[25] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. "SCONE: Secure Linux Containers with Intel SGX". *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[26] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. "VC3: Trustworthy Data Analytics in the Cloud Using SGX". *2015 IEEE Symposium on Security and Privacy*.

[27] E. Bauman, H. Wang, M. Zhang, and Z. Lin. "SGXElide: Enabling Enclave Code Secrecy via Self-Modification". *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*.

[28] T. Lazard, J. Götzfried, T. Müller, G. Santinelli, and V. Lefebvre. "TEEshift: Protecting Code Confidentiality by Selectively Shifting Functions into TEEs". *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18)*.

[29] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. "OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX". *Proceedings 2019 Network and Distributed System Security Symposium (NDSS '19)*.

[30] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni. "Twine: An Embedded Trusted Runtime for WebAssembly". *2021 IEEE 37th International Conference on Data Engineering (ICDE)*.

[31] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza. "TrustJS: Trusted Client-Side Execution of JavaScript". *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*.

[32] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza. "AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting". *Proceedings of the 20th International Middleware Conference (Middleware '19)*.

[33] A. Baumann, M. Peinado, and G. Hunt. "Shielding Applications from an Untrusted Cloud with Haven". *ACM Trans. Comput. Syst.* (2015).

[34] NVIDIA. *NVIDIA H100 Tensor Core GPU Architecture*. `https://resources.nvidia.com/en-us-tensor-core`.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. "Efficient Software-Based Fault Isolation". *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*.

[36] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization". *2015 IEEE Symposium on Security and Privacy*.

[37] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*.

[38] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual. Volume 2: System Programming*. Accessed December 2022.

[39] ARM. *Arm Architecture Reference Manual for A-profile architecture*. `https://developer.arm.com/documentation/ddi0487`.

[40] N. Fenton and M. Neil. "A critique of software defect prediction models". *IEEE Transactions on Software Engineering* (1999).

[41] *CVE security vulnerability database. Security vulnerabilities, exploits, references and more*. `https://www.cvedetails.com`.

[42] Advanced Micro Devices Inc. *AMD Secure Encrypted Virtualization (SEV)*. `https://developer.amd.com/sev/`. Accessed: January 2020.

[43] V. Costan and S. Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086.

[44] *XuCode: An Innovative Technology for Implementing Complex Instruction Flows*. `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html`. Version published on 06 May 2021.

[45] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. *Intel Software Guard Extensions: EPID Provisioning and Attestation Services*.

[46] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski. *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives*.

[47] Advanced Micro Devices Inc. *SEV Secure Nested Paging Firmware ABI Specification*. `https://www.amd.com/system/files/TechDocs/56860.pdf`. Revision: 1.54.

[48] S. Pinto and N. Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". *ACM Computing Surveys* (2019).

[49] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base". *22nd USENIX Security Symposium (USENIX Security '13)*.

[50] V. Costan, I. Lebedev, and S. Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation". *25th USENIX Security Symposium (USENIX Security '16)*.

[51] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. "Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software". *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[52] A. Moghimi, G. Irazoqui, and T. Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". *Cryptographic Hardware and Embedded Systems – CHES 2017*.

[53] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. "Cache Attacks on Intel SGX". *Proceedings of the 10th European Workshop on Systems Security (EuroSec '17)*.

[54] S. P. Johnson. *Intel SGX and Side-Channels*. `https://software.intel.com/content/www/us/en/develop/articles/intel-sgx-and-side-channels.html`. Accessed May 2020.

[55] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". *Proceedings 2017 Network and Distributed System Security Symposium (NDSS '17)*.

[56] Intel Corporation. *Protection from Side-Channel Attacks*. `https://software.intel.com/content/www/us/en/develop/documentation/sgx-developer-guide/top/protection-from-sidechannel-attacks.html`. Accessed May 2020.

[57] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*.

[58] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li. "Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX". *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019).

[59] J. Van Bulck, F. Piessens, and R. Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*.

[60] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd. "Mitigating Branch-Shadowing Attacks on Intel SGX Using Control Flow Randomization". *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18)*.

[61] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". *29th USENIX Security Symposium (USENIX Security '20)*.

[62] A. Limited. *mbedTLS (formerly known as PolarSSL)*. https://tls.mbed.org/. Accessed March 2020.

[63] Intel Corporation. *Cryptography for Intel Integrated Performance Primitives Developer Reference*. https://software.intel.com/content/www/us/en/develop/documentation/ipp-crypto-reference/top.html. Accessed October 2020.

[64] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors". *2009 30th IEEE Symposium on Security and Privacy*.

[65] Intel Corporation. *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations*. https://software.intel.com/security-software-guidance/insights/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations. Accessed March 2020.

[66] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution". *2019 IEEE Symposium on Security and Privacy*.

[67] J. Van Bulck, F. Piessens, and R. Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX'17)*.

[68] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs*. https://www.agner.org/optimize/microarchitecture.pdf. Accessed: January 2020.

[69] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. "DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries". *27th USENIX Security Symposium (USENIX Security '18)*.

[70] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar. "MicroWalk: A Framework for Finding Side Channels in Binaries". *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*.

[71] C. D. Walter and S. Thompson. "Distinguishing Exponent Digits by Observing Modular Subtractions". *Topics in Cryptology — CT-RSA 2001*.

[72] A. Cabrera Aldaya and B. B. Brumley. "When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA". *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020).

[73] O. Acıiçmez, S. Gueron, and J.-P. Seifert. "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures". *Cryptography and Coding 2007*.

[74] A. Cabrera Aldaya, A. J. C. Sarmiento, and S. Sánchez-Solano. "SPA vulnerabilities of the binary extended Euclidean algorithm". *Journal of Cryptographic Engineering* (2017).

[75] B. Krzanich. *Advancing Security at the Silicon Level*. `https://newsroom.intel.com/editorials/advancing-security-silicon-level/#gs.yh984y`. Accessed March 2020.

[76] B. Solomon, A. Mendelson, R. Ronen, D. Orenstien, and Y. Almog. "Micro-operation cache: A power aware frontend for variable instruction length ISA". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2003).

[77] A. Rane, C. Lin, and M. Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution". *24th USENIX Security Symposium (USENIX Security '15)*.

[78] M. Hähnel, W. Cui, and M. Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". *2017 USENIX Annual Technical Conference (USENIX ATC '17)*.

[79] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". *27th USENIX Security Symposium (USENIX Security '18)*.

[80] Y. Yarom, D. Genkin, and N. Heninger. "CacheBleed: a timing attack on OpenSSL constant-time RSA". *Journal of Cryptographic Engineering* (2017).

[81] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar. "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations". *International Journal of Parallel Programming* (2019).

[82] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri. "Port Contention for Fun and Profit". *2019 IEEE Symposium on Security and Privacy*.

[83] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. "MicroScope: Enabling Microarchitectural Replay Attacks". *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*.

[84] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx. "Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution". *Engineering Secure Software and Systems (2018)*.

[85] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. "Preventing Page Faults from Telling Your Secrets". *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*.

[86] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. “SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs”. *Proceedings 2017 Network and Distributed System Security Symposium (NDSS ’17)*.

[87] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. “KASLR is Dead: Long Live KASLR”. *Engineering Secure Software and Systems (2017)*.

[88] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. *Journal of Cryptographic Engineering* (2018).

[89] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX”. *26th USENIX Security Symposium (USENIX Security ’17)*.

[90] R. Guanciale, H. Nemati, C. Baumann, and M. Dam. “Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures”. *2016 IEEE Symposium on Security and Privacy*.

[91] O. Acıiçmez. “Yet Another MicroArchitectural Attack: Exploiting I-Cache”. *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW ’07)*.

[92] D. A. Osvik, A. Shamir, and E. Tromer. “Cache Attacks and Countermeasures: The Case of AES”. *Topics in Cryptology – CT-RSA 2006*.

[93] O. Acıiçmez and W. Schindler. “A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL”. *Topics in Cryptology – CT-RSA 2008*.

[94] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. “Cross-VM Side Channels and Their Use to Extract Private Keys”. *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS ’12)*.

[95] E. Tromer, D. A. Osvik, and A. Shamir. “Efficient Cache Attacks on AES, and Countermeasures”. *Journal of Cryptology* (2010).

[96] Y. Yarom and K. Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. *23rd USENIX Security Symposium (USENIX Security ’14)*.

[97] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA ’16)*.

[98] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA ’17)*.

[99] *Confidential Computing concepts | Confidential VM*. https://cloud.google.com/compute/confidential-vm/docs/about-cvm. Accessed: August 2022.

[100] *Microsoft Docs: Build with SGX enclaves - Azure Virtual Machines*. https://docs.microsoft.com/en-us/azure/confidential-computing/confidential-computing-enclaves. Accessed: August 2022.

[101] *Azure Confidential VM options on AMD*. https://docs.microsoft.com/en-us/azure/confidential-computing/virtual-machine-solutions-amd. Accessed: August 2022.

[102] *Awesome WebAssembly Languages*. https://github.com/appcypher/awesome-wasm-langs. Accessed: August 2022.

[103] *WebAssembly Micro Runtime*. https://github.com/bytecodealliance/wasm-micro-runtime.

[104] *Perft Test Benchmarks for crates.io/chess/, crates.io/shakmaty/*. https://crates.io/crates/chess. Version: 3.1.1.

[105] *Wasmtime: A standalone runtime for WebAssembly*. https://github.com/bytecodealliance/wasmtime.

[106] *Wasmer - The Universal WebAssembly Runtime*. https://wasmer.io/.

[107] A. Abel and J. Reineke. "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures". *ASPLOS (ASPLOS '19)*.

[108] I. Puddu, M. Schneider, M. Haller, and S. Capkun. "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend". *30th USENIX Security Symposium (USENIX Security 21)*.

[109] *Official WebAssembly test suite*. https://github.com/WebAssembly/spec/tree/main/test.

[110] T. K. Ho. "Random Decision Forests". *Proceedings of 3rd International Conference on Document Analysis and Recognition*. IEEE.

[111] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi. "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures". *Proceedings 2020 Network and Distributed System Security Symposium (NDSS)*.

[112] *Intel Architecture Instruction Set Extensions Programming Reference*. https://cdrdv2.intel.com/v1/dl/getContent/671368. Accessed December 2022.

[113] B. Parno. "Bootstrapping Trust in a Trusted Platform". *HotSec'08*.

[114] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer. "Physical Key Extraction Attacks on PCs". *Commun. ACM* (2016).

[115] Z. Wang and R. Lee. “Covert and Side Channels Due to Processor Architecture”. *2006 22nd Annual Computer Security Applications Conference (ACSAC’06)*.

[116] K. Gandolfi, C. Mourtel, and F. Olivier. “Electromagnetic Analysis: Concrete Results”. *Cryptographic Hardware and Embedded Systems — CHES 2001*.

[117] D. Genkin, A. Shamir, and E. Tromer. “Acoustic Cryptanalysis”. *Journal of Cryptology* (2016).

[118] S. Brands and D. Chaum. “Distance-Bounding Protocols”. *Advances in Cryptology — EUROCRYPT ’93*.

[119] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. “Meltdown: Reading Kernel Memory from User Space”. *27th USENIX Security Symposium (USENIX Security ’18)*.

[120] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. *27th USENIX Security Symposium (USENIX Security ’18)*.

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

[122] S. Sasy, S. Gorbunov, and C. W. Fletcher. “ZeroTrace : Oblivious Memory Primitives from Intel SGX”. *Proceedings 2018 Network and Distributed System Security Symposium*.

[123] D. Wendlandt, D. G. Andersen, and A. Perrig. “Perspectives: Improving SSH-style Host Authentication with Multi-path Probing”. *2008 USENIX Annual Technical Conference (USENIX ATC ’08)*.

[124] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. “ROTE: Rollback Protection for Trusted Execution”. *Proceedings of the 26th USENIX Conference on Security Symposium (SEC’17)*.

[125] S. Agarwal. *Public Cloud Inter-region Network Latency as Heat-maps*.

[126] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. “Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics”.

[127] I. Algo-Logic Systems. *Low Latency PCIe Solutions for FPGA*. `https://www.algo-logic.com/sites/default/files/PCIe.pdf`. 2019.

[128] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS ’19)*.

[129] *Tiny Core Linux, Micro Core Linux, 12MB Linux GUI Desktop, Live, Frugal, Extendable*. https://distro.ibiblio.org/tinycorelinux/. 2018.

[130] Z. Zhang, X. Ding, G. Tsudik, J. Cui, and Z. Li. "Presence Attestation: The Missing Link in Dynamic Trust Bootstrapping". *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.

[131] R. A. Fink, A. T. Sherman, A. O. Mitchell, and D. C. Challener. "Catching the Cuckoo: Verifying TPM Proximity Using a Quote Timing Side-Channel". *Trust and Trustworthy Computing*.

[132] S. P. Johnson, U. R. Savagaonkar, V. R. Scarlata, F. X. McKeen, and C. V. Rozas. *Technique for supporting multiple secure enclaves*. US Patent 8,972,746.

[133] F. X. McKeen, C. V. Rozas, U. R. Savagaonkar, S. P. Johnson, V. Scarlata, M. A. Goldsmith, E. Brickell, J. T. Li, H. C. Herbert, P. Dewan, et al. *Method and apparatus to provide secure application execution*. US Patent 9,087,200.

[134] L. Chen. *SP 800-108. Recommendation for Key Derivation Using Pseudorandom Functions (Revised)*. Tech. rep. 2009.

[135] M. J. Dworkin. "Recommendation for block cipher modes of operation: The CMAC mode for authentication" (2016).

[136] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. "Rethinking Software Runtimes for Disaggregated Memory". *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*.

[137] C. Percival. *Cache missing for fun and profit*. `http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf`. 2005.

[138] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation". *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*.

[139] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks". *Information Security and Cryptology - ICISC 2005*.

[140] C. Liu, M. Hicks, and E. Shi. "Memory Trace Oblivious Program Execution". *2013 IEEE 26th Computer Security Foundations Symposium*.

List of Figures

2.1 Example of a typical hardware configuration on a modern computing architecture. The edges between the components show the communication topology. Blue (dark) edges represent typical internal communication links, while the yellow (light) ones show how external entities are connected. 14

2.2 Software organization on the system 16

2.3 Privilege levels and expected software to be running on each level (according to [37, 39]) on x86 (left) and ARM (right). Note that numbers decrease (increase) with more privileges in x86 (ARM). 19

2.4 4-level virtual address translation lookups performed on an x86 CPU to translate a virtual address into a physical one. The tables contain only physical addresses, and the virtual address is used as an index to these tables. The CPU performs this translation in hardware every time a virtual address is accessed. 24

2.5 Context switch steps performed on an x86 CPU to support multitasking. 25

2.6 Total reported CVEs (irrespective of severity) by year for three popular Operating Systems: Linux, Windows 10, and Mac OS X. Data from [41]. The figure also depicts a linear regression model fit over the data points for each OS to highlight the trend over the years. 27

2.7 Isolation boundaries and software allocation for TEEs across three architectures: Intel x86, AMD x86, and ARM. The figure shows both a baseline without protection and four TEEs: Intel SGX, AMD SEV-SNP, ARM TrustZone, and ARM CCA. Each block in the figure represents an isolated software module. Vertical separation is achieved through privilege level mechanisms (cf. Section 2.2.1), while horizontal separation is enforced for lower levels by software executing on the level above. Double lines (|| and |¦) only apply to ARM. Particularly, |¦ applies to ARM TrustZone, and represents one-way isolation: secure word software can access the normal word memory, but the opposite is not possible. || applies only to ARM CCA and represents full isolation: Realms cannot access the rest of the system, and the rest of the system cannot access the Realms. 28
2.8 Life cycle of an Intel SGX enclave. The OS and the untrusted host application are depicted in red, as they are not trusted in the SGX attacker model. Some instructions require privilege level (PL) 0 to be executed and can thus be done only by the OS. The enclave executes with the privileges of a user-level application (PL = 3). 31

3.1 A secret-dependent branch in C and x86 assembly. Both branches in the assembly code fit within the same cacheline (64B). The virtual address of the instructions is reported on the left. Note that while the branches are instruction-wise identical, their instructions get grouped differently by the fetch window (which always starts at multiples of 16B). . . 47
3.2 Distribution of the overall execution time of the branches in Listing 3.2 when run outside of SGX without interrupts (computed from $2 * 10^9$ samples). 51
3.3 Attack success rate depending on the alignment of the branches. The attack success rate is the percentage of correctly guessed branches by the attacker out of 1000 executed branches. The 10th instruction (5th `mov`) from Listing 3.2 is used to distinguish between both branches. The color gradient goes from darker to brighter, where darker boxes indicate higher attack success rates (up to 100%) and brighter ones lower success rates (down to 50%). 54

3.4 Timing distribution of a `mov` to the stack when executing it in a trace containing $100,000$ repeated `add-mov` instructions (unrolled). 56
3.5 Timing distribution of the `movs` from Figure 3.4 grouped by their virtual address alignment. 58
3.6 Timing distributions of two different `movs` in the IPP Cryptography library's `l9_ippsCmp_BN` function (each estimated from 3000 samples). The function executes a secret dependent comparison, which can result in two balanced paths being taken: the *bigger-than* or *smaller-than* path. Each path contains a differently-aligned `mov` in it, whose distribution is shown in the figure. 61
3.7 Comparison of the real subtraction (*if* branch) and dummy subtraction (*else* branch) branches in the mbedTLS MM implementation. The two branches are identical, and both include a `for loop` that executes two memory writes (cf. Listing A.1). The graph shows the distribution of the 11th instruction in the `for loop` (a `reg` to `reg` *subtraction*), highlighting that as long as memory writes are present, surrounding loop instructions produce different distributions based on their alignment as well. The distributions were estimated from 1000 function calls, each of which has 6 loop iterations, resulting in 6000 measurements per instruction. 63
3.8 Distribution of the attack success rate with different microcode versions of an Intel Core i7-7700 CPU – across 500 runs per microcode. For each run, we estimate the attack success rate as the percentage of branches the attacker guessed correctly among 1000 executed branches from Listing 3.2, with alignment $X = 6, Y = 2$. 66

4.1 The two main approaches providing code confidentiality with TEEs: native execution (above the dashed line) and IR execution (below the dashed line). 76
4.2 The two approaches to code confidentiality in TEEs. The native execution enclave (left) gets the source code compiled to x86; the IR execution enclave (right) gets as input WASM bytecode. Both systems operate in an environment with a malicious OS and are tasked with executing the same source code. 80

4.3 Sample trace collection during the execution of an x86 `imul` and one of its WASM equivalents, `i32.mul`. The x86 instruction generates a single Instruction Measurement (IM), while the WASM instruction generates 9 IMs due to executing 9 underlying x86 instructions. 81

4.4 Instruction candidate set size distribution of semantically different SGX and SEV instructions for various 64-bit x86 microarchitectures under the SotA attacker. The plot shows the minimum candidate set size that contains at least x percent of the ISA available in the TEE (SEV for AMD and SGX for Intel). This assumes the best resolution available to the SotA attacker with respect to execution time is 10 cycles. The dotted red line is set at $y = 10$, and it indicates that $> 90\%$ of the ISA instructions have a candidate set size greater than 10. 88

4.5 Instruction candidate set size distribution of semantically different SGX and SEV instructions for various 64-bit x86 microarchitectures under the ideal attacker. The dotted green line is set at $y = 1$, and given where it intersects the various microarchitectures' ISA, it indicates that more than 90% of the instructions cannot be recovered even by the ideal attacker. 89

4.6 Candidate set size distribution of WASM instructions in the WAMR interpreter under the SotA attacker. Only semantically different instructions are included in the candidate sets. The green dotted line is at $y = 1$, where candidate sets of that size offer no confidentiality. 93

4.7 Overview of the end-to-end attack steps. 95

4.8 Timing distribution of the 5th x86 instruction for the five listed WASM instructions. The two division operations seem to be following a different distribution than the others. N=11527 100

4.9 Confusion matrix of a simple random forest classifier for five WASM instructions. The classifier is pretty confident about the two divisions but cannot distinguish the other 3 instructions. 102

5.1 Differences when performing (a) an AEX without AEX-Notify and (b) with AEX-Notify enabled and implemented. 110

6.1 Relay attack: The adversary redirects attestation to their own platform, which gives them increased (side channel and kernel level) abilities to attack the attested enclave. . . 121

6.2 Relay attack implications: The tree shows the types of attacks that are enabled by redirection and ones that are independent of relay. 122

6.3 Example sequences of events. In Case A, the attack success is independent of relay. In Case B, attestation redirection enables the attack. 124

6.4 PROXIMITEE attestation: The remote verifier establishes a secure channel to the PROXIMIKEY device that first attests the enclave and then verifies its proximity. 129

6.5 Sliding window: for periodic proximity verification with three different types of challenge-response latencies. 130

6.6 Our experimental setup: consists of the PROXIMIKEY device prototype, the target platform, the attacker's platform, and the connection interfaces between them. 135

6.7 Latency distributions: for legitimate challenge-response rounds (left) and simulated relay attack (right). 137

6.8 Parameter tuning: the attacker's success probability P_{adv} and the legitimate success probability P_{legit} for different numbers of rounds n given a fixed k. 138

6.9 Boot-time initialization: The PROXIMIKEY uses a minimal kernel Linux image to boot and load the PROXIMITEE enclave on the target platform and seal a platform-specific secret to the PROXIMIKEY memory. 142

6.10 PROXIMITEE boot-time attestation: After the boot-time initialization (refer to Figure 6.9), the PROXIMITEE enclave executes a local attestation with the verifier uploaded application-specific enclave. 143

7.1 Figure from [135] depicting the two cases for CMAC computation. The case on the left occurs if the message length is a multiple of the block size; otherwise, the case on the right is used. 152

7.2 Representation of the steps used to recover a block message M_i from an AES-CMAC tag. The blue arrows are computed from the first message block towards the middle, iterating over the blue message blocks with equation (7.1). The red arrows are computed from the end (the MAC tag) towards the middle, iterating over the red message blocks with equation (7.3). By combining these two steps, we obtain M_i. 153

A.1 Distribution of the instructions of Figure 3.5 across different runs, split by their alignments. This figure highlights how different enclave runs exhibit a shift in the mean of their instructions' distribution, and hence distributions are not directly comparable between runs. 171

B.1 Candidate set sizes' distributions on the Skylake microarchitecture for a SotA attacker in the native system with varying cycle accuracy thresholds. 176

List of Tables

3.1 View of how instructions are batched into fetch windows when the enclave resumes execution, according to which branch is executing. If an instruction crosses a fetch window boundary, we assume it is decoded together with the instructions in the following window. The interrupts refer to the instructions in Figure 3.1b. 49

3.2 List of all the processors we tested with their respective microcode versions. The *Mitig.* column indicates whether the mitigation against known microarchitectural attacks such as Spectre and Foreshadow is implemented in hardware (HW) or μcode. 65

3.3 Overview and comparison of related SGX side-channel attacks. The first two columns indicate whether the attack can leak data-dependent or control-flow (CF) dependent secrets. The Frontal attack is the only attack that can leak the decision made for any type of branch (as long as they contain a memory store in them), even if they are based on indirect unconditional jumps (e.g., as a mitigation against BPU attacks), or if both paths are contained within the same CL (e.g., as a mitigation against cache and controlled-channel attacks). 69

4.1 View of the attacker for the `asm` in Listing 4.2; $y = 2$. Candidate sets contain only semantically different instructions. Collected in the Skylake microarchitecture. . . 87

4.2 Attacker view of the loop in Listing 4.2 when the source code is compiled to WASM. We report the number of executed IMs recorded both when loading (as done in the first JIT phase) and interpreting each WASM instruction. Some instructions are simplified by the JIT loader and are thus not present in the interpreter trace. Instructions with a bold font are executed both in the loading and interpreting phase, while the other instructions only during loading. Numbers are computed from the same version of WAMR as in Listing 4.1. Compared to Table 4.1, here we report only one iteration of the loop (due to space constraints). The second iteration would see the bold instructions repeated. Candidate sets contain only semantically different WASM instructions. . . . 92

7.1 Summary of notation 151

8.1 Values that influence the `VCEK` key. Taken from [47, p. 18] . 158

List of Listings

3.1 Protection against timing attacks in the latest version (v2.16.6 at the time of writing) of MbedTLS. The library balances branches by having symmetric execution paths. . . 42
3.2 ASM Code with high attack success probability, which we use to profile the attack. The `.rept 25 ... .endr` assembler directive repeats the instructions within the block 25 times, leading to an address of `x+0x190` for the `ret` instruction. . . 50

4.1 Excerpt of the main loop of the Bytecode alliance WAMR interpreter [103] (commit `b554a9d`) responsible for loading a WASM binary. `opcode` (line 5) is the opcode of the current WASM instruction being parsed. This listing shows how a control-flow dependency on the `opcode` usually manifests (line 8) in WASM interpreters and compilers, and how different instructions exhibit different amplification factors. For instance, `WASM_OP_IF` (line 14) requires multiple operations to be translated, amplifying the information available to the attacker compared to the equivalent functionality in x86 (usually a single instruction). 83
4.2 A simple assembly program with a loop that, on each iteration, computes $z = z * x$. The loop iterates y times. The variable x is stored on `%eax`, y on `-8(%rbp)`, and z on `%ecx`. 86

A.1 Exploited `for` loop in the mbedTLS library's `mpi_montmul` function (compiled on `gcc 7.5.0` with `-O3`). 172
A.2 Exploited `for` loop in the `mbedtls_rsa_gen_key` function of the mbedTLS library. 173