unine
Université de Neuchâtel
Institut d'informatique

# Enhancing Security and Performance in Trusted Execution Environments

**Thèse**

**Présentée à la Faculté des sciences de l'Université de Neuchâtel**

**Pour l'obtention du grade de Docteur ès Sciences**

**par**

**Peterson Yuhala**

Soutenu le 16 février 2024

Université de Neuchâtel
Faculté des sciences

Rue Emile-Argand 11
CH-2000 Neuchâtel
doctorat.sciences@unine.ch

# IMPRIMATUR POUR THESE DE DOCTORAT

**La Faculté des sciences de l'Université de Neuchâtel autorise l'impression de la présente thèse soutenue par**

**Monsieur Peterson YUHALA**

Titre :

**"Enhancing Security and Performance in Trusted Execution Environments"**

**sur le rapport des membres du jury composé comme suit :**

- Prof. Pascal Felber, directeur de thèse, Université de Neuchâtel, Suisse
- Dr Valerio Schiavoni, Université de Neuchâtel, Suisse
- Prof. Alain Tchana, Grenoble INP, France
- Prof. Leonardo Querzoni, Università degli studi di Roma 1, Italie
- Prof. Fernando Pedone, Università della Svizzera italiana USI, Suisse

Neuchâtel, le 22 avril 2024

Le Doyen, Prof. R. Bshary

[illegible]

*À Gertrude et Fred Racine.*

# ABSTRACT

Cloud computing permits companies to easily scale their infrastructure to meet changing demand, providing reduced costs and improved scalability. However, for cloud tenants, ensuring the confidentiality and integrity of sensitive data entrusted to untrusted cloud service providers poses security challenges. To address these issues, processor manufacturers have introduced *trusted execution environments* (TEEs) like Intel software guard extensions (SGX), AMD secure encrypted virtualisation (SEV), Arm TrustZone *etc*. into commodity CPUs. TEEs provide secure *enclaves* that leverage hardware-based security mechanisms to ensure the confidentiality and integrity of code and data deployed on untrusted cloud infrastructures.

Nevertheless, the complexity of TEEs at the development level has been a major impediment to their widespread adoption. Two fundamental challenges arise when dealing with TEE programs: improving security through code partitioning, and mitigating the performance overhead introduced by the security extensions. While some tools have been proposed by both academia and industry to address these issues, they have drawbacks: they either target only lower level programming languages, hindering adoption by the development community, or are too language-specific, limiting their utility for programs in different programming languages.

This thesis revisits these fundamental issues in TEE development and presents more viable solutions. First, it addresses the code partitioning challenge by designing and building high-level, as well as multi-language tools, to isolate sensitive code inside secure enclaves. Second, it explores the realm of IoT, providing robust techniques to enhance security in IoT environments. Finally, it introduces novel approaches to tackle the performance challenges associated with TEEs.

The first research work presents an approach to partitioning Java programs for Intel SGX enclaves. The proposed approach relies on code annotations and bytecode transformations to partition Java classes into trusted and untrusted components. These partitioned components are then ahead-of-time (AOT) compiled into binaries that run in and out of an enclave, while maintaining sufficient interaction to preserve the functional goals of the original program.

Building upon the previous work, the second research work extends the concept of code partitioning to a multi-language context. It leverages GraalVM's Truffle framework to provide abstract syntax tree (AST) nodes that encapsulate sensitive data in polyglot programs. The resulting AST is then analysed via dynamic taint analysis, and the secure nodes are used to deduce sensitive portions of the program to be isolated inside an enclave.

The third research work delves into the realm of internet of things (IoT), providing a technique to secure sensitive data generated by peripheral devices. It proposes a generic blueprint for partitioning peripheral drivers for Arm TrustZone, and leverages this approach to design and build a robust framework to enhance security of IoT peripherals.

The fourth research work shifts the focus to performance enhancement by leveraging persistent memory (PM) to provide efficient fault tolerance guarantees for Intel SGX programs. The practicality of this technique is demonstrated in a machine learning (ML) context to achieve secure and persistent ML model training.

Still focusing on performance enhancement, the fifth research work proposes a system to optimise

Intel SGX switchless calls. It identifies the limitations of the static configuration policy in Intel SGX's switchless call library, and provides a dynamic approach which obviates the performance penalty due to static configurations.

**Keywords:** security and privacy, trusted execution environments, Intel SGX, Arm TrustZone, cloud computing, GraalVM, Truffle framework, persistent memory.

# RÉSUMÉ

Le cloud offre une flexibilité permettant aux entreprises de s'adapter rapidement aux fluctuations de la demande, accordant ainsi des coûts réduits et une meilleure scalabilité. Cependant, pour les utilisateurs du cloud, garantir la confidentialité et l'intégrité des données sensibles confiées à des fournisseurs de services cloud non fiables pose des défis en matière de sécurité. Pour résoudre ce problème, les fabricants de processeurs ont introduit des *environnements d'exécution de confiance* (TEE) tels que Intel software guard extensions (SGX), AMD secure encrypted virtualisation (SEV), Arm TrustZone, *etc*., dans les processeurs. Les TEEs fournissent des *enclaves* sécurisées qui exploitent des mécanismes de sécurité basés sur le matériel pour garantir la confidentialité et l'intégrité du code et des données déployés sur des infrastructures cloud non fiables.

Cependant, la complexité des TEEs au niveau du développement a été un obstacle majeur à leur adoption par les développeurs. Deux défis fondamentaux se posent lors du développement des programmes TEE : améliorer la sécurité grâce au partitionnement du code et atténuer les surcoûts de performance introduits par les extensions de sécurité. Bien que certaines solutions aient été proposées par le milieu académique et l'industrie pour résoudre ces problèmes, elles présentent des inconvénients : elles ciblent soit uniquement des langages de programmation de bas niveau, entravant l'adoption par la communauté du développement, soit sont trop spécifiques à un langage particulier, limitant leur utilité pour les programmes basés sur différentes langages de programmation.

Cette thèse revisite ces problèmes fondamentaux dans le développement TEE et présente des solutions plus viables. Tout d'abord, elle aborde le défi du partitionnement du code en concevant et en développant des outils de haut niveau, ainsi que des outils multi-langages, pour isoler le code sensible à l'intérieur d'enclaves sécurisées. Deuxièmement, elle explore le domaine de l'internet des objets (IoT), en fournissant des techniques robustes pour renforcer la sécurité dans les environnements IoT. Enfin, elle propose des approches nouvelles pour relever les défis de performance associés aux TEEs.

Le premier travail de recherche introduit une nouvelle approche pour partitionner les programmes Java pour les enclaves Intel SGX. L'approche proposée s'appuie sur des annotations de code et des transformations de bytecode pour partitionner les classes Java en composants fiables et non fiables. Ces composants partitionnés font l'objet d'une compilation anticipée (AOT) permettant ainsi leur exécution à l'intérieur et à l'extérieur d'une enclave, tout en maintenant une interaction suffisante pour préserver les objectifs fonctionnels du programme original.

S'appuyant sur le travail précédent, le deuxième travail de recherche étend le concept de partitionnement du code à un contexte multi-langage. Il exploite le framework Truffle de GraalVM pour fournir des nœuds d'arbre syntaxique abstrait (AST) qui encapsulent des données sensibles dans des programmes polyglottes. L'AST résultant est ensuite analysé via une analyse dynamique, et les nœuds sécurisés sont utilisés pour déduire les portions sensibles du programme à isoler à l'intérieur d'une enclave.

Le troisième travail de recherche explore le domaine de l'internet des objets (IoT), en proposant une technique pour sécuriser les données sensibles générées par les périphériques. Il fournit un modèle générique pour le partitionnement des pilotes de périphériques, et exploite cette approche pour concevoir et construire un framework robuste pour renforcer la sécurité des périphériques IoT en utilisant Arm TrustZone.

Le quatrième travail de recherche recentre l'attention sur l'amélioration des performances en utilisant la mémoire persistante (PM) pour fournir des garanties efficaces de tolérance aux pannes pour les programmes Intel SGX. La praticabilité de cette technique est démontrée dans un contexte d'apprentissage automatique (ML) pour réaliser un entraînement de modèle ML sécurisé et persistant.

Se focalisant toujours sur l'amélioration des performances, le cinquième travail de recherche propose un système pour optimiser les appels sans commutation ("switchless calls") d'Intel SGX. Il identifie les limitations de la politique de configuration statique de la librairie switchless d'Intel SGX, et propose une approche dynamique qui élimine la pénalité de performance due aux configurations statiques.

**Mots-clés :** sécurité et confidentialité, environnements d'exécution de confiance, Intel SGX, Arm TrustZone, cloud computing, GraalVM, Truffle, mémoire persistante.

# ACKNOWLEDGMENTS

This dissertation is the culmination of years of hard work and dedication, and I owe a debt of gratitude to many people who have supported me along the way.

First and foremost, I want to express my profound gratitude to my supervisors, Prof. Pascal Felber, Dr. Valerio Schiavoni, and Prof. Alain Tchana, for their unwavering support and tutelage throughout my tenure as a PhD student. Pascal, I am immensely grateful for the opportunity to pursue my PhD under your supervision. Your insights, feedback, experience, and immense knowledge have been invaluable in shaping my research. I am equally grateful for the freedom you granted me during the course of my research, which enabled me make independent decisions, contributing significantly to the success and innovation of my work. Valerio, thank you for your extensive mentorship and guidance, which played a crucial role in helping me achieve this significant milestone in my academic career. Your impact on my academic journey will undoubtedly resonate throughout my career. Alain, I am very grateful for the internship opportunity that eventually led to my PhD. Your unwavering commitment to excellence has challenged me to push myself beyond my limits and to constantly strive for improvement.

I wish to express my deep appreciation to the external members of my thesis committee, Prof. Leonardo Querzoni and Prof. Fernando Pedone for their constructive comments and suggestions that significantly improved the quality of this dissertation.

I owe special thanks to Hugo Guiroux, Jean-Pierre Lozi, and Prof. Gaël Thomas, with whom I collaborated on some research projects. Your expertise and guidance were indispensable in overcoming various technical challenges.

I would like to express my heartfelt gratitude to my family. First and foremost, thank you Mum and Dad, for instilling in me the love for education and a deep appreciation for the value of learning, which have been the foundation of my academic success. I would never have achieved this goal without you. I am equally grateful to my siblings Nadia, Kathya, Julia, and Darren whose love and encouragement have been instrumental in keeping me motivated and focused throughout this challenging journey.

Je tiens à remercier Gertrude et Fred Racine, pour m'avoir acceuilli en Suisse avec tant de bienveillance. Votre hospitalité et gentillesse ont fait de cette expérience un moment de vie exceptionnel et inoubliable. Je remercie également mes chers amis Cecilia, Mohammed, Ricardo, Catherine, Laurence, et toute l'équipe des Crêts pour les beaux moments passés ensembles. E para ti, Thaïs, eras uma pessoa maravilhosa. Que encontres descanso nos braços dos anjos.

I would like to acknowledge the contributions of all my colleagues and friends at IIUN, who have provided a supportive and stimulating academic and work environment. Merci à mon tonton adoptif, Rafael Pires pour m'avoir assisté dans mes débuts avec Intel SGX, et pour les astuces en programmation C: now I have much smaller `main` functions and use lots of C macros. Thanks to Sebastien Vaucher for being a walking encyclopedia, ready to impart knowledge on so many subjects ranging from Swiss laws to SGX microcode updates. Thanks to Remi Dulong for the knowledge exchanged while working together on persistent memory. I'm happy you left behind a large office for me alone, so I could finally croon in peace. Thank you Dorian Burihabwa for the gym tips, and for always being ready to help. Thank you Christian Goëttel for your selfless devotion to saloon administration. Thank you Jämes Ménétrey for all your assistance while working on GraalVM and TEEs; we make such a wonderful team! Thank you Baptiste Lepers for your willingness to proof-read my papers and offer very constructive feedback. I look forward to having a DBLP profile as intimidating as yours. Thanks Gal for being a great hiking partner; I will surely find my way to Haifa someday.

It is important to achieve a harmonious equilibrium between lab work and social life. For this, I am profoundly grateful to Rafael Pires, Sebastien Vaucher, and Dorian Burihabwa. Thank you guys for the weekend invites, hiking trips around Switzerland, and the beer exploits where I "felt my eyes".

I would also take this opportunity to thank my former colleagues in France: Stella Bitchebe, Djob Mvondo, Kevin Jiokeng, and all past and present members of the IRIT lab, with whom I spent valuable time remotely during the course of my PhD. You have been a crucial part of my PhD journey.

Finally, I am grateful to everyone who has helped me with my work both in word and deed, whom I am not able to mention here by name.

Peterson Yuhala
Neuchâtel, January 5, 2024

[illegible]

# PREFACE

This thesis presents research conducted at the Complex Systems lab of the Faculty of Sciences at the University of Neuchâtel, Switzerland, to obtain a PhD in Computer Science.

The research activities were supervised primarily by Prof. Pascal Felber (University of Neuchâtel), Dr. Valerio Schiavoni (University of Neuchâtel), and Prof. Alain Tchana (Grenoble INP). Some of the works presented in this thesis were done in collaboration with Oracle Labs Zürich.

This thesis provides a diversity of research works with the overarching theme of improving both security and performance within trusted execution environments.

The research conducted led to five publications at international conferences:

- Plinius: Secure and Persistent Machine Learning Model Training [223] (IEEE/IFIP International Conference on Dependable Systems and Networks 2021)
- Montsalvat: Intel SGX Shielding for GraalVM Native Images [225] (ACM/IFIP International Middleware Conference 2021)
- SecV: Secure Code Partitioning via Multi-Language Secure Values [222] (ACM/IFIP International Middleware Conference 2023)
- SGX Switchless Calls Made Configless [221] (IEEE/IFIP International Conference on Dependable Systems and Networks 2023)
- Fortress: Securing IoT Peripherals with Trusted Execution Environments [224] (ACM/SIGAPP Symposium on Applied Computing 2024)

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACRONYMS

| | |
|---|---|
| OS | operating system |
| VAS | virtual address space |
| SRAM | static random-access memory |
| DRAM | dynamic random-access memory |
| TLB | translation lookaside buffer |
| SGX | Intel software guard extensions |
| TDX | Intel trust domain extensions |
| SEV | AMD secure encrypted virtualisation |
| TEE | trusted execution environment |
| EDL | enclave definition language |
| EPC | enclave page cache |
| EPCM | enclave page cache map |
| SECS | SGX enclave control structure |
| LE | launch enclave |
| TCS | thread control structure |
| EDMM | enclave dynamic memory management |
| SVN | security version number |
| SVN | software version number |
| ISV | independent software vendor |
| MEE | memory encryption engine |
| MMU | memory management unit |
| TLB | translation lookaside buffer |
| AEX | asynchronous exit |
| SSA | state save area |
| MC | memory controller |
| PRM | processor reserved memory |
| MAC | message authentication code |
| TCB | trusted computing base |
| TZASC | TrustZone address space controller |
| TZ | Arm TrustZone |
| SMC | secure monitor call |
| SCR | secure configuration register |
| PM | persistent memory |
| NVM | non-volatile memory |
| PMDK | persistent memory development kit |
| SSD | solid state drive |
| HDD | hard disk drive |
| DAX | direct access |
| IO | input output |
| VM | virtual machine |
| JVM | Java virtual machine |
| JIT | just-in-time |
| AOT | ahead-of-time |
| AST | abstract syntax tree |
| JVMCI | Java virtual machine compiler interface |
| GC | garbage collector |
| ML | machine learning |
| IoT | internet of things |

CHAPTER 1

# Introduction

This chapter aims to contextualise the research work done, so as to provide a clearer understanding of the purpose and rationale of this PhD thesis. It presents an overview of the key contributions made, and provides a general outline for the rest of the manuscript.

The chapter is organised as follows:

## 1.1 CONTEXT AND MOTIVATION

In recent years companies have moved their computing workloads from on-premises to public clouds, which permit them to easily scale their infrastructure to meet changing demand. Instead of investing in costly hardware and software, companies can simply outsource their computations to cloud service providers, and pay for the computing resources they need. This flexibility provides reduced costs, improved availability, and reliability.

However, the prevalence of cloud services has introduced new security challenges [124, 168]. On the one hand, when potentially sensitive data is entrusted to an untrusted cloud provider, there is a risk of data exposure or tampering [11, 161, 164, 228] by the cloud service provider, *e.g.*, via a rogue administrator with superuser privileges. On the other hand, in a multi-tenant cloud environment where multiple customers share the underlying hardware and software infrastructure through virtualisation, vulnerabilities in privileged software such as the hypervisor [35, 67, 229] could lead to unauthorised access to private data by a neighbouring compute instance, or other unauthorised entities.

Hence, cloud computing presents a new threat model wherein some elements of the computing platform, *i.e.*, hardware (*e.g.*, memory) or software (*e.g.*, operating system, hypervisor) fall within the adversary's control. This means traditional process isolation by the operating system (OS) and virtualisation techniques are insufficient to guarantee security. These security issues represent the core motivation behind the *confidential computing* [124] paradigm.

Confidential computing aims to bridge the gap between the convenience of cloud computing and the paramount need for data security and privacy. It provides a computing approach that ensures data remains confidential even when processed in cloud infrastructures where the underlying privileged software stack, *i.e.*, OS, hypervisor is considered untrusted. This is achieved by introducing software isolation primitives at the processor level, significantly reducing the number and size of components that need to be trusted by a cloud client. Major CPU manufacturers like Intel, Arm, and AMD have implemented confidential computing by introducing *trusted execution environments* (TEEs) like Intel software guard extensions (SGX) [34], Intel trust domain extensions (TDX) [22], Arm TrustZone [159], SEV [37], and RISC-V MultiZone Security [173] into their processor technologies. TEEs leverage hardware primitives to provide encrypted memory regions called *enclaves* which provide robust confidentiality and integrity guarantees to applications, even in the presence of potentially malicious system software.

***Challenge 1: code partitioning for better security***

To enhance security, the TEE must maintain a small trusted computing base. This can be achieved via *program partitioning*, which reduces the potential attack surface. However, manual partitioning is usually complex and error-prone, and existing automatic partitioning tools focus primarily on lower level languages.

**Challenge 1.** Despite the strong security guarantees provided by TEEs, integrating TEE capabilities into an application poses some difficulties. Firstly, the APIs provided are typically implemented in lower-level programming languages like C, C++, or assembly, making development difficult for non-experts. Secondly, most TEEs impose a number of restrictions (motivated by security) on application code, *e.g.*, disallowing system calls [195]; this requires the application code to be adapted to leverage the security features provided by the TEE. This lack of flexibility at the language level coupled with the restrictions on application code has led to the emergence of tools such as *library OSs* [7, 11, 160, 175, 195] that enable unmodified applications to run inside an enclave, while providing in-enclave system services [61] like user-level scheduling, in-memory filesystem, *etc*. While these tools simplify

TEE-based development by offering a higher level of abstraction to the programmer, they introduce a new problem: naively including large amounts of code inside a TEE bloats the *trusted computing base (TCB)*,[1] posing a security risk. This is because any vulnerabilities present in the library OS (LibOS) or the in-enclave program can be leveraged by an adversarial party, leading to a security breach. Moreover, from an empirical point of view, there is a strong correlation between the number of security vulnerabilities in a given code base and the size and complexity of the code base. To put this into perspective, a standard Linux environment comprises over 10 million lines of code. This not only represents a huge attack surface that could be compromised by an attacker, but further complicates formal verification [35, 90, 129] techniques to validate the correctness of the system.

This problem can be mitigated through *privilege separation* [15, 104], a security principle which divides a program into parts with different levels of privilege or access rights. Several hardware architectures provide isolation mechanisms to achieve privilege separation, *e.g.*, between kernel and user mode: rings 0 and 3 in Intel x86 [198], exception levels 1 and 0 in ARM [102], and S- and U-modes in RISC-V [207]. With regards to TEE development, this principle can be extended to user space software by protecting only sensitive code and data in the TEE. This is achieved through ***program partitioning***, which involves splitting the code base into two subprograms: a *trusted component* which executes inside the TEE, and an *untrusted component* which executes out of the TEE. These components interact and collectively implement the original program, but the system as a whole now fulfills the important security requirement: restricting access to sensitive information within the enclave. Partitioning significantly reduces the attack surface for the most security-critical components, and often results in improved performance [226, 222] as the untrusted component does not incur any performance penalties due to the TEE. Several runtime libraries have been developed that allow developers to ***manually partition*** enclave code (mostly written in C or C++) into trusted and untrusted components, as well as handle any possible interactions between the partitioned components. These include: Intel SGX SDK [30], Microsoft OpenEnclave SDK [130], Google Asylo [54], Fortanix's Rust Enclave Development Platform [43], and Keystone SDK [85] for RISC-V architectures.

While re-designing a given program this way often achieves improved security and performance goals, establishing and maintaining the trusted-untrusted boundary effectively requires careful consideration. Moreover, ensuring that sensitive operations and data are confined to the secure side while allowing appropriate interactions with non-secure components can be complex, especially when dealing with large code bases. These difficulties are further compounded when dealing with high-level programming languages like Java, JavaScript, Python *etc*. for which the TEE implementations provide little or no (direct) support. As an illustrative example, partitioning a Java program for a TEE requires handling references and dependencies between objects residing in the trusted and untrusted components, which operate on separate heaps. Manual handling of these complexities introduces serious engineering challenges, underscoring the necessity for ***automatic or semi-automatic partitioning*** tools to abstract away the complexity.

Academia and industry have proposed some automatic program partitioning tools, but unfortunately the proposed tools usually target lower-level programming languages like C or C++ [104, 177] or legacy frameworks [170, 14]. This is insufficient because a huge percentage of cloud-based applications and frameworks [44, 45, 93] are implemented in high-level languages like Java, Python, JavaScript, *etc*. While there have been efforts [196] to address this disparity, they still introduce large language runtimes like the Java virtual machine (JVM) and a library OS [195] inside the TEE. There is thus a need to develop higher-level language frameworks and APIs that do not only simplify TEE programming, but also provide better TCB reduction guarantees.

1 The TCB refers to the set of software, firmware, and hardware components that are critical to the security of a computer system.

***Challenge 2: improving performance of TEEs***

TEEs introduce significant overhead, *e.g.*, through frequent context switches to perform unsupported operations like I/O. Current solutions to this problem have limitations, highlighting the demand for more viable performance enhancement techniques.

**Challenge 2.** The security guarantees provided by TEEs often come at a cost. Firstly, popular TEE implementations like Intel SGX enclaves operate only in user mode. As a result, enclave programs are required to perform complex operations such as CPU context switches to access OS related APIs like system calls (*i.e.*, `read`, `write`, *etc.*) which are ubiquitous in modern programs. These additional operations introduce substantial performance overhead. For instance, invoking a non-enclave function from within an SGX enclave results in an overhead ranging from 8000 to 14, 000 CPU cycles, depending on cache state [210, 157]. This represents a performance degradation of up to 93× when compared to a regular OS system call, which incurs about 150 CPU cycles. Although prior research endeavours [187, 157, 7] have proposed solutions to mitigate this performance problem, there is still room for further enhancement. As an example, the Intel SGX software development kit provides a tool to statically configure transitionless cross-enclave function invocations using additional worker threads. However, these static configurations are difficult and misconfigurations can result in performance degradations and waste of CPU resources.

Secondly, the security restrictions imposed by TEE technologies like Intel SGX require software changes, *e.g.*, in the C standard library. While the Intel SGX SDK provides the required reimplementations, some are suboptimal and could be improved.

Lastly, TEE technologies typically provide limited memory resources for secure programs. To illustrate, the initial version of Intel SGX introduced on the Skylake microarchitecture provides a maximum of $128\,MB$ of secure memory, of which only $\approx 93.5\,MB$ is usable by enclaves. Once this memory capacity is exceeded, expensive paging operations [182, 34] are triggered to swap enclave pages to regular DRAM. While the enclave memory budget has been increased on recent server-grade processors, a large number of SGX-capable processors still grapple with the aforementioned memory limitation. This limitation has performance implications, especially for applications requiring substantial memory or I/O operations (*e.g.*, `read`, `fread`, *etc.*) relying on large buffers in secure memory. These challenges significantly constrain the practicality of adopting TEEs for large-scale computations, such as machine learning training algorithms.

The prevalence of such performance-related issues has led to a growing demand for techniques that can enhance the performance of TEE-based programs while maintaining acceptable security guarantees.

## 1.2 CONTRIBUTIONS

This thesis aims, on the one hand, to unlock the potential of TEE technologies for high-level programming languages, by providing generic tools and abstractions that distill the underlying complexity inherent in TEE development. On the other hand, it addresses the performance challenges associated with TEEs by introducing novel ideas or improving existing ones.

With regards to the language aspect, technologies like GraalVM [13] and Truffle [212, 153] provide a rich software ecosystem to enhance interoperability between relatively low-level languages like C/C++, and more popular high-level languages *e.g.*, Java,
JavaScript, Python, *etc.* We leverage these two technologies to design and build automatic TEE code

partitioning tools for these popular high-level languages, thereby easing adoption of TEE technologies for non-experts.

On the performance side, novel memory technologies like PM are a promising solution to mitigate the high I/O costs of programs. PM has both storage and memory characteristics, and provides memory access semantics that can be leveraged by TEEs to eschew costly CPU context switches required to manipulate data in the underlying storage device. We leverage these properties of PM to provide efficient fault tolerance guarantees for in-enclave data structures. Further, we investigate how multithreading can be efficiently leveraged in TEEs to provide optimal performance.

In summary, the contributions of this thesis are as follows.

**Program partitioning tool for Java.** We propose a tool, MONTSALVAT, that leverages code annotations as well as bytecode transformations to partition Java programs for Intel SGX enclaves. The partitioned components are then AOT compiled by GRAALVM into binaries that execute in and out of an enclave in a distributed fashion.
This work was done in collaboration with Hugo Guiroux, Jean-Pierre Lozi (Oracle labs Zürich), and Jämes Ménétrey (University of Neuchâtel). It was supervised by Pascal Felber, Valerio Schiavoni (University of Neuchâtel), Alain Tchana (Grenoble-INP), and Gaël Thomas (Télécom SudParis). MONTSALVAT was published in the 22nd ACM/IFIP International Middleware Conference 2022 [225], and constitutes Chapter 3 of this thesis.

**Multi-language program partitioning tool.** We design and implement SECV, a multi-language tool for partitioning applications in a wide range of high-level programming languages such as JavaScript, Python, amongst others, for Intel SGX enclaves. SECV leverages GraalVM's Truffle framework to provide AST nodes that encapsulate sensitive data in polyglot programs. The programs are then dynamically analysed and partitioned based on these nodes.
SECV was realised in collaboration with Hugo Guiroux and Jean-Pierre Lozi (Oracle labs Zürich), and was supervised by Pascal Felber, Valerio Schiavoni (University of Neuchâtel), Alain Tchana (Grenoble-INP), and Gaël Thomas (Télécom SudParis). It was published in the 24th ACM/IFIP International Middleware Conference 2023 [222] and constitutes Chapter 4 of this thesis.

**Securing IoT Peripherals.** We delve into the realm of IoT, where we provide a generic blueprint for partitioning peripheral drivers for Arm TrustZone. We apply this approach to shield audio peripherals in an IoT setup.
This work was done in collaboration with Jämes Ménétrey (University of Neuchâtel), and supervised by Pascal Felber, Marcelo Pasin, and Valerio Schiavoni (University of Neuchâtel). It was published in the 39th ACM Symposium on Applied Computing 2024 [224], and constitutes Chapter 5 of this thesis.

**Efficient fault tolerance guarantees for enclaves using PM.** To address the performance issues in TEEs, we propose a novel technique which leverages persistent memory to provide efficient fault tolerance guarantees for TEE based applications. Our contribution comprises a *mirroring mechanism* which involves creating encrypted mirror copies of enclave data structures on PM, and synchronising the enclave and PM copies throughout the program's lifetime. We demonstrate the feasibility and efficiency of this approach by applying it to machine learning model training.
This work was supervised by Pascal Felber, Valerio Schiavoni (University of Neuchâtel) and Alain Tchana (ENS Lyon). It was published in the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2021 [223], and constitutes Chapter 6 of this thesis.

**Configless Intel SGX switchless calls.** Still within the context of TEE performance improvement, we design and implement ZC-SWITCHLESS, a configless approach for Intel SGX switchless calls. ZC-SWITCHLESS identifies drawbacks in Intel's static configuration policy for switchless calls and provides a more dynamic approach that minimises waste of CPU resources and obviates the performance penalty due to poor static configurations. The results of this work were obtained in collaboration with Michael Paper (ENS de Lyon) and Timothée Zerbib (Institut Polytechnique de Paris). It was supervised by Pascal Felber, Valerio Schiavoni (University of Neuchâtel) and Alain Tchana (Grenoble-INP). ZC-SWITCHLESS was published in the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2023 [221], and constitutes Chapter 7 of this thesis.

Two additional articles [183, 116] were published during the course of this thesis. The subjects addressed in these publications are outside of the scope of the material covered here and thus do not form part of this manuscript. The complete bibliographic reference of each of the publications is listed starting from page 123.

## 1.3 SOFTWARE ARTIFACTS

The tools outlined in the contributions above have been openly released to the scientific community.

- Chapter 3: `https://github.com/Yuhala/montsalvat.git`
- Chapter 4: `https://gitlab.com/Yuhala/generic-tools.git`
- Chapter 6: `https://github.com/Yuhala/plinius.git` and `https://github.com/Yuhala/sgx-romulus.git`
- Chapter 7: `https://gitlab.com/Yuhala/zc-switchless.git`

## 1.4 THESIS OUTLINE

This manuscript is organised as follows:

Chapter 2 provides background information on the key technologies leveraged in this thesis: TEEs, GraalVM, and PM. We then organise the manuscript into two parts:

Part I focuses on designing and implementing code partitioning tools for TEEs. We first introduce MONTSALVAT in Chapter 3, for partitioning programs written in a high-level programming language like Java for Intel SGX. In Chapter 4, we then introduce SECV, our multi-language approach for partitioning programs for Intel SGX. We round up Part I in Chapter 5, where we present FORTRESS, a robust system to shield IoT peripherals with Arm TrustZone.

Part II shifts the focus to TEE performance improvement. In Chapter 6, we present a novel approach to provide efficient fault-tolerance guarantees in enclave programs by leveraging PM. Subsequently in Chapter 7, we propose optimisations to Intel's switchless calls, an approach which mitigates expensive enclave context switches. Chapter 8 concludes this thesis and offers insights into what lies ahead.

CHAPTER 2

# Background

In this chapter we present key concepts and frameworks that underpin this work. These include: trusted execution environments, GraalVM, and persistent memory (PM).

The chapter is organised as follows:

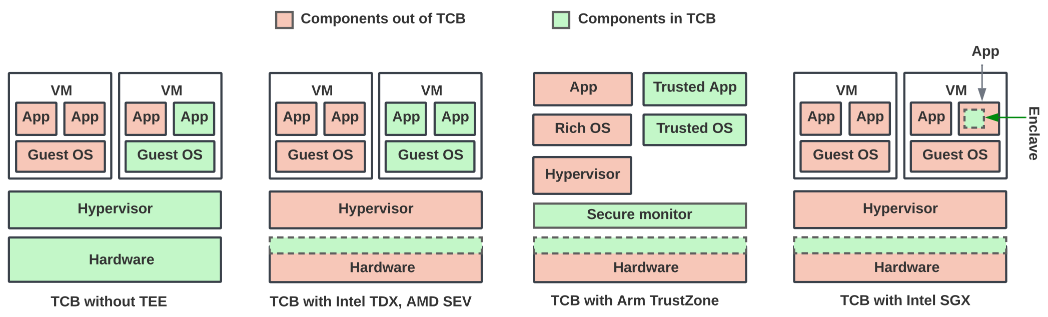


**Figure 2.1:** Trusted computing base with and without TEEs.

## 2.1 TRUSTED EXECUTION ENVIRONMENTS

Contemporary computer programs deal with various forms of sensitive data, *e.g.*, encryption keys, passwords, health or financial data, *etc*. This data could exist in three states: *at rest*, *e.g.*, on a hard drive (locally or in the cloud), *in transit*, *e.g.*, a password being transferred across the network from a web browser to a server, or *in use*,[1] *e.g.*, an encryption key being used to decrypt sensitive data in a process. Up until the past decade, the primary emphasis of security was on safeguarding data at rest or in transit. However, with the rise of cloud computing, concerns regarding the security of data in use have come to the forefront [124].

In the past, cloud clients had to fully trust the cloud service provider's compute infrastructure, *i.e.*, the OS, hypervisor, and hardware when performing computations involving sensitive data. This model posed significant security risks. Moreover, a malicious OS or hypervisor, which was beyond the control of the end user, could potentially compromise the application. Over the years, a variety of techniques have been developed to mitigate these security concerns. One of such techniques is *fully homomorphic encryption* [41, 63, 201], which enables computations directly on encrypted data, but incurs substantial performance penalties. Alternatively, *trusted platform modules* (TPMs) [87] enable the verification of the host platform's integrity at system initialisation, but can't protect applications against attacks at runtime. To tackle this challenge, popular hardware vendors like Intel, Arm, and AMD introduced the concept of a *trusted execution environment* (TEE) [18], a more efficient approach to ensure security guarantees for data in use.

A trusted execution environment (TEE) is an isolated processing environment provided by a processor to ensure the ***confidentiality***, ***integrity***, and ***freshness*** of a process's code and data. *Confidentility* ensures that unauthorised or malicious individuals, entities, or processes cannot access code or data secured by the TEE. *Integrity* ensures that the secured code or data cannot be modified in an unauthorised or undetected manner. *Freshness* ensures the most up-to-date version of the information being secured is always read.

Several TEE implementations exist: *Intel Software Guard Extensions* (SGX) [34, 31], *Intel Trust Domain Extensions* (TDX) [22], *Arm TrustZone* [159], *AMD Secure Encrypted Virtualisation* (SEV) [37], and RISC-V Keystone [94]. These TEE implementations are tailored for specific use cases and address distinct threat models. Figure 2.1 depicts the trust model for a legacy computing stack versus that with modern TEE implementations. In a legacy stack, *i.e.*, without TEEs, the entire computing stack including the hardware, hypervisor, and OS need to be trusted. In contrast, TEE implementations like AMD SEV and Intel TDX isolate an entire virtual machine (VM), while excluding the hypervisor and a significant portion of the hardware from the TCB. Similarly, Arm TrustZone excludes the hypervisor and a large part of the hardware from the trust boundary, but provides the possibility of including

1 Data in memory and CPU registers.

a small trusted OS in the TCB. A recent Arm-based TEE technology for protecting virtual machines, the confidential compute architecture (CCA) [99] has been introduced into Armv9 architecture that further shrinks the trust boundary when compared to TrustZone. Finally, Intel SGX provides the smallest TCB, including only the processor and a small portion of the application, the enclave.

In this thesis, our primary focus is on Intel SGX and Arm TrustZone. The former is the most widely used TEE implementation in cloud environments [119, 26, 83], while the latter is the dominant technology in ARM-based user-end devices [159].

**Table 2.1:** Intel SGX instructions.

| **Instruction** | **Description** |
|---|---|
| *Supervisor mode* | |
| `ECREATE` | Creates an enclave by converting a free EPC page to a SECS |
| `EADD` | Adds a page to an enclave |
| `EREMOVE` | Removes a page from the EPC |
| `EEXTEND` | Generates a cryptographic hash of enclave content in 256*B* chunks |
| `EINIT` | Initialises an enclave |
| `EBLOCK` | Blocks all accesses to a page being prepared for eviction |
| `ELDB` | Loads an evicted page into the EPC as blocked |
| `ELDU` | Loads an evicted page into the EPC as unblocked |
| `ETRACK` | Tracks logical processors that have flushed TLBs |
| `EWB` | Invalidates an EPC page and copies it to main memory |
| `EPA` | Creates a version array to store a nonce generated by `EWB` |
| `EDBGRD` | Reads any page in a debug enclave |
| `EDBGWR` | Writes data to a debug enclave |
| *User mode* | |
| `EENTER` | Enters an enclave |
| `EEXIT` | Exits an enclave |
| `ERESUME` | Resumes enclave execution after an asynchronous exit (AEX) |
| `EGETKEY` | Derives CPU-unique cryptographic keys |
| `EREPORT` | Creates a cryptographic report used for attestation |

### *2.1.1 Intel Software Guard Extensions*

Intel SGX is an extension to the Intel instruction set architecture [174] first introduced in the Skylake generation of Intel x86 CPUs. It permits user space applications to create trusted execution environments called *enclaves*. SGX enclaves are secure isolated regions in the application's virtual address space (VAS) which leverage hardware-based mechanisms to provide confidentiality and integrity guarantees to application code and data [112]. The list of CPU instructions that constitutes SGX is summarised in Table 2.1.

**Enclave memory.** SGX enclave code and data reside in a secure memory region called the *enclave page cache (EPC)*. The EPC is part of *processor reserved memory (PRM)*, a contiguous portion of dynamic random-access memory (DRAM) which is inaccessible to non-enclave software, including system software. EPC memory is split into 4KB pages which can be assigned to different enclaves. The CPU tracks the metadata of each EPC page in a micro-architectural structure called the *enclave page cache map (EPCM)*. The latter is a look-up table within the CPU package which holds one entry per EPC page. Each EPCM entry stores page access permissions (*i.e.*, read, write, execute), an identifier for the owner enclave, and the mapped virtual address [61, 34] for the corresponding EPC page. The EPCM

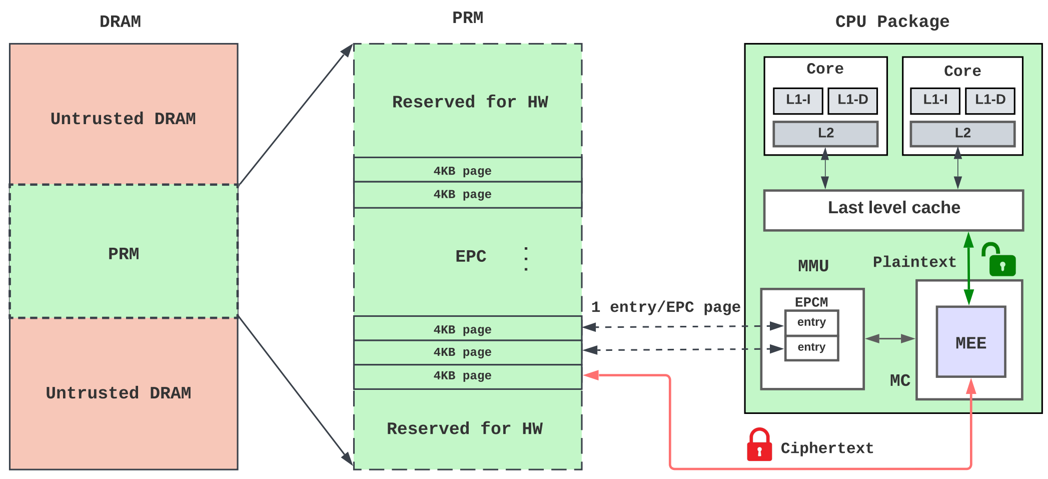


**Figure 2.2:** Intel SGX memory architecture.

structure is used by the memory management unit (MMU) for access control checks during address translation for enclave virtual addresses.

SGX stores metadata for each enclave in a data structure called the *SGX enclave control structure (SECS)*. SGX instructions use the virtual address of the SECS as an identifier for the corresponding enclave. Each SECS is stored in a dedicated EPC page.

Furthermore, SGX stores metadata for each enclave thread in a thread control structure (TCS). The TCS contains information about the thread's state, such as register values and the entry point for trusted code to be executed by a logical processor. This information is used each time a logical processor performs a context switch to and from enclave mode. Similar to the SECS, each TCS consumes a single EPC page [34, 112].

**Enclave memory confidentiality and integrity.** In the Intel SGX adversary model, system memory as well as the memory bus is susceptible to snooping and malicious tampering. An extension of the memory controller (MC) called the *memory encryption engine (MEE)* ensures the confidentiality, integrity, and replay attack protection for EPC pages in DRAM [62]. As illustrated in Figure 2.2, all CPU read/write requests corresponding to addresses in the PRM are routed by the MC to the MEE which either transparently encrypts the data from the CPU cache line before it is written to DRAM, or decrypts data from the EPC when it is loaded into a CPU cache line. The associated cryptographic key used for these operations is generated by a hardware random number generator during system boot, and is accessible only to the MEE.

To ensure data integrity and freshness, the MEE computes a message authentication code (MAC) tag over each data block (*i.e.*, a 64*B* CPU cache line) and an associated *nonce*, a non-repeating number used only once for each cryptographic operation on a data unit. The MAC tag is attached to the ecrypted data block being written to DRAM, while the nonce is stored in an integrity tree. The latter is implemented as a Merkle Tree [62, 34] where each node contains a MAC tag used to verify its children, and leaf nodes contain MAC tags used to verify the integrity of actual cachelines written to EPC memory. The nodes of the integrity tree are stored in PRM (outside the EPC), except for the root node, which is stored in CPU-internal static random-access memory (SRAM) which is inaccessible to software, thereby making the tree tamper resistant. Figure 2.3 provides a visual representation of a four-ary MEE integrity tree.

When data is being loaded from the EPC into a CPU cache line, the MEE uses the MAC tags and nonces to perform integrity (and freshness) checks to ensure the data has not been tampered with (or

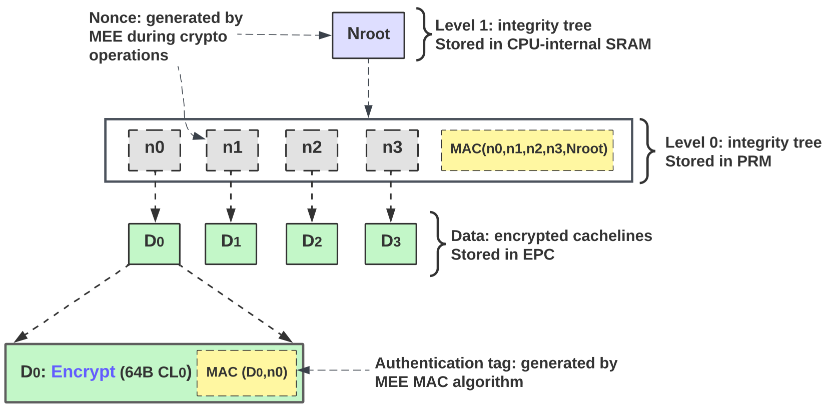

Nonce: generated by MEE during crypto operations
Nroot
Level 1: integrity tree Stored in CPU-internal SRAM
n0
n1
n2
n3
MAC(n0,n1,n2,n3,Nroot)
Level 0: integrity tree Stored in PRM
D0
D1
D2
D3
Data: encrypted cachelines Stored in EPC
D0: Encrypt (64B CL0)
MAC (D0,n0)
Authentication tag: generated by MEE MAC algorithm

**Figure 2.3:** Four-ary Intel SGX MEE integrity tree.

replayed). Any mismatch during these verifications will cause the MEE to emit a failure signal which immediately locks the MC, thereby requiring a system reboot, after which new keys are generated for the MEE's operation [62]. Data integrity verifications by the MEE are the primary reason for the performance degradation observed when reading/writing EPC pages, as well as the limited EPC budget (up to 256$MB$) on most SGX systems. However, recent third-generation Intel Xeon scalable processors support up to 512$GB$ of EPC memory per CPU. This increase in capacity, though, involves a compromise: the omission of memory integrity and replay protection [80].

**EPC paging.** Intel CPUs from the SkyLake to Gemini Lake (exclusive) generation support a maximum of 128 MB of EPC memory, of which only 93.5 MB is usable by SGX enclaves. This severely limits the total amount of code and data that can be accomodated by all enclaves on a system. To mitigate this problem, the Linux SGX kernel driver provides a paging mechanism that permits the swapping of pages between the EPC and regular DRAM. This enables enclave applications to use more memory than provided by the EPC, but at a significant cost [182, 14].

For SGX-enabled CPU generations prior to Gemini Lake, an enclave's set of committed memory pages together with their access permissions (*i.e.*, read, write, execute) is fixed at enclave load time. As such, the size of the enclave's stack or heap cannot be changed at runtime. On more recent Gemini Lake based Intel processors, *e.g.*, Intel Celeron J4005 Processor,[2] new CPU instructions have been added to the original SGX instruction set that introduce the possibility to add more pages to enclave memory dynamically, and are together referred to as SGX2, or *enclave dynamic memory management (EDMM)*. SGX2 instructions introduce the possibility to perform several memory management operations in enclaves such as: on-demand heap/stack growth, page permission modifications, dynamic module/library loading, on-demand creation of code pages for just-in-time (JIT) compilation, *etc*. [111, 214]. We refer to the original SGX instruction set as SGX1. Unless stated otherwise, all Intel SGX-related work in this thesis has been done with SGX1-based systems.

**Enclave execution flow.** An SGX enclave program is subdivided into two parts: *trusted part* and *untrusted part*. The trusted part represents the enclave code itself, and contains all the code that operates on sensitive data. The untrusted part on the other hand contains all code that does not operate on sensitive data at runtime. It is built as a regular executable file, while the trusted part is built as a *shared object* (`.so`) on Linux systems, or *dynamic-link library* (`.dll`) on Windows systems.

Figure 2.4 describes the execution flow of an Intel SGX program. Firstly, the untrusted component of the program is run, with its `main` function as the entry point. The `ECREATE` instruction is then used to

2 Intel Celeron J4005 Processor

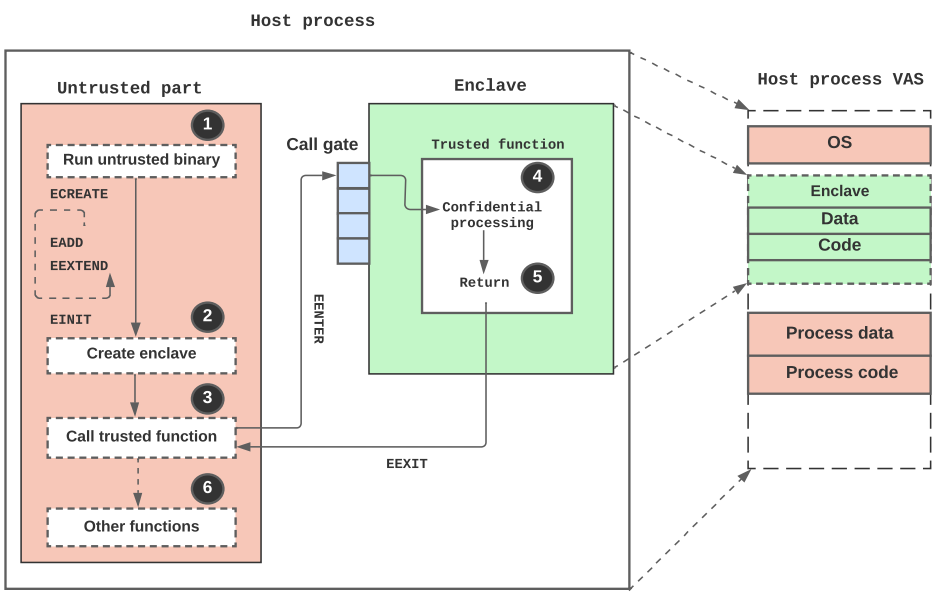


**Figure 2.4:** Intel SGX program execution flow.

create an enclave in the host process's VAS. ECREATE essentially turns a free EPC page into a SECS which contains attributes such as the base address and size of the enclave. Once the SECS is created, the system software uses the EADD instruction to add $4KB$ pages containing code or data into the enclave, and creates TCS pages for enclave threads. As these pages are being loaded into the enclave, the EEXTEND instruction is used to generate a cryptographic hash of the content of enclave pages in $256B$ chunks. Once the initial code and data is loaded, system software leverages a special SGX architectural enclave called the *launch enclave (LE)* to obtain an initialisation token that is provided to the EINIT instruction, which completes the initialisation step of the enclave once executed.

Upon enclave initialisation, its code can be executed by the corresponding host process by issuing the EENTER instruction. The latter takes a TCS address as parameter, saves the address of the following instruction (in the host process) in the RBX register, and switches the logical processor into *enclave mode*. It then performs a controlled jump into the enclave code's trusted function via a predefined *entry point* (*i.e.*, call gate).

Once execution of the trusted function terminates (*i.e.*, returns), the EEXIT instruction is triggered. EEXIT triggers a TLB flush, marks the corresponding TCS as free, and transfers control back to the host process at the saved RBX address.

If a hardware exception, *e.g.*, a fault or interrupt, occurs while a logical processor is in enclave mode, the processor performs an AEX. The latter saves information about the enclave's state (*e.g.*, stack pointer) in a data structure called a state save area (SSA), invokes the exception handler, and eventually resumes enclave execution via the ERESUME instruction.

**Enclave signing and attestation.** The security of Intel SGX based applications hinges on a mandatory *enclave signing* step. This process provides a unique signature that identifies all software within the enclave, as well as the signing authority, *e.g.*, the enclave developer. Once an enclave program is compiled, a secure hash digest of its initial code and data, called the enclave *measurement* (a 256-bit hash) is created. This value is stored in the signed enclave file within a *signature structure*: SIGSTRUCT. Before the enclave is loaded into EPC memory at runtime, the CPU recalculates the enclave's measurement and stores the value in a special register: MRENCLAVE. The CPU then compares this value to that in the SIGSTRUCT obtained at build time [56, 34]. These values need to match for

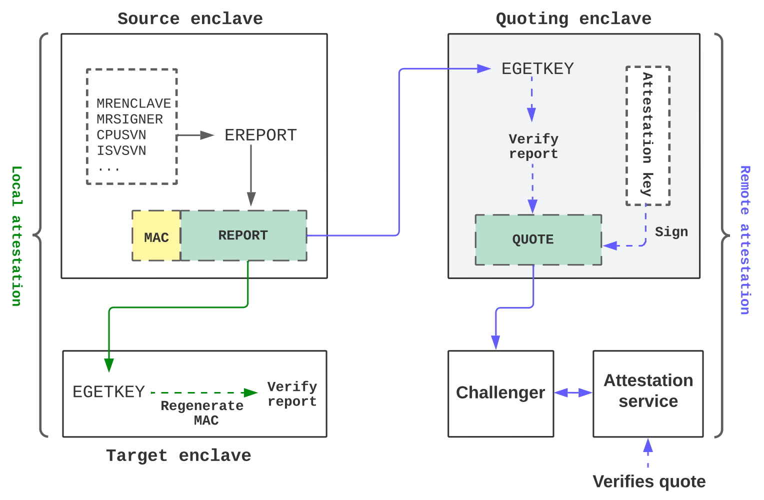


**Figure 2.5:** Intel SGX local and remote attestation.

the enclave to load successfully, otherwise the load is aborted.

The `SIGSTRUCT` data structure also contains an identifier for the enclave signing entity, which is essentially a hash of the enclave author's public key. After an enclave is successfully loaded in the EPC and initialised, the CPU stores the value of the signing entity's hash identifier in another special register: `MRSIGNER`. This value is used during enclave attestation.

Similar to enclave signing, enclave *attestation* is a crucial component in the deployment of Intel SGX based programs, and permits cryptographic verification of the integrity and authenticity of an SGX enclave. Intel SGX provides two attestation techniques:

***Local attestation.*** This is a mechanism used by an enclave (*i.e.*, *source enclave*) to prove its identity (and authenticity) to another enclave (*i.e.*, *target enclave*) hosted by the same SGX-enable CPU. SGX provides an instruction: `EREPORT`, which is used by the source enclave to generate a cryptographic structure known as an *attestation report* (*i.e.*, `REPORT` [34]). The report contains the values of `MRENCLAVE` and `MRSIGNER`, CPU information, *e.g.*, software version number (SVN), and independent software vendor (ISV) information. All this information is signed to produce a MAC tag that is attached to the attestation report. The report is then sent to the target enclave which verifies its authenticity by leveraging another SGX instruction: `EGETKEY`. The latter obtains the cryptographic key used to generate the attached MAC tag, which permits to verify the authenticity of the source enclave [34].

***Remote attestation.*** This is a mechanism that allows a remote party (*i.e.*, a cloud client or "challenger") to ensure that their deployed software has not been tampered with, and is running within an enclave on a system with an acceptable security level, *i.e.*, CPU microcode version. The SGX remote attestation mechanism leverages a privileged system enclave called the *quoting enclave* to verify a given enclave, *i.e.*, source enclave. This enclave performs a local attestation with the quoting enclave, which then verifies the local attestation report provided by the source enclave, and cryptographically signs this report with a CPU-unique attestation key to produce an attestation called a *quote*. This quote is sent to an *Intel attestation service* [81, 89] which verifies that it was indeed produced on an SGX-enabled system. Intel SGX local and remote attestation mechanisms are summarised in Figure 2.5.

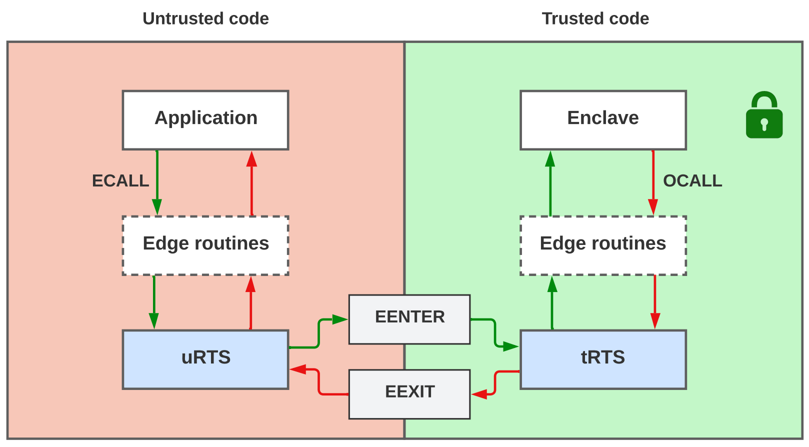


**Figure 2.6:** Enclave interface showing edge routines.

### *2.1.2 Intel SGX software development kit (SDK)*

Intel provides an SDK [30] comprising various tools and libraries to facilitate the development and deployment of Intel SGX applications.

**Enclave development framework.** The SGX SDK provides a C/C++ based enclave development framework that provides a generic SGX application structure which allows developers to manually partition their code into trusted and untrusted parts, to be run in and out of the enclave respectively. To enable communication across both partitions, the SDK provides a set of programming interfaces: `ecall` and `ocall` routines. An `ecall` (*enclave call*) is a specialised function invoked from the untrusted partition of an SGX application to enter an enclave. An `ecall` performs an `EENTER` operation (see Figure 2.4) which essentially performs a controlled jump into a trusted enclave function (*e.g.*, to process a secret). An `ocall` (*out call*) on the other hand is a specialised function invoked from within the enclave to execute an external function that resides in the untrusted partition. An `ocall` performs an `EEXIT` operation. Ocalls are commonly used by the enclave to access system calls or perform input output (IO) operations which cannot be done in enclave mode.

The `ecall/ocall` interface is defined via an *enclave definition language (EDL)* in specialised `.edl` files. The SGX SDK provides a tool called *Edger8r* [30] (pronounced "*edgerator*") which automatically generates *edge routines* using the EDL specification. As illustrated in Figure 2.6, edge routines are functions that run outside the enclave (untrusted edge routines) or inside the enclave (trusted edge routines) and serve to bind a call from the application with a function inside the enclave or a call from the enclave with a function in the application. Edge routines are equally used to properly sanitise and marshal input parameters passed between the application and the enclave via `ecall` or `ocall` routines. For example, an EDL syntax like: `public void ecall_set_key([in, size=4]void* ptr)` declares an ecall with a data pointer as parameter. The `[in, size=4]` attribute directs the corresponding edge routine to allocate a 4-byte memory buffer inside the enclave and copy 4 bytes of data from the untrusted part of the application into the allocated buffer inside the enclave once the `ecall` is invoked. This ensures enclave memory is not unintentionally (or maliciously) overwritten when data is copied into the enclave during the `ecall`.

**Trusted and untrusted runtime systems.** The Intel SGX SDK provides a support framework comprising the *trusted runtime system (tRTS)* and the *untrusted runtime system (uRTS)*, to facilitate the interaction between the enclave and the untrusted application. More specifically, the tRTS provides tools to manage the enclave itself, perform `ocalls`, and receive `ecalls`. Similarly the uRTS performs functions

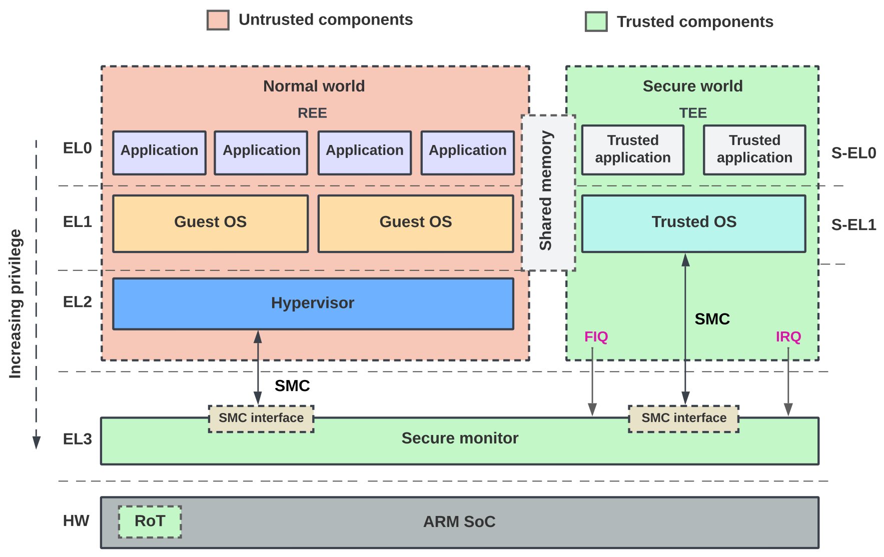


**Figure 2.7:** TrustZone security architecture on ARM Cortex-A.

such as loading the enclave, performing `ecalls`, and receiving `ocalls`.

**Trusted C standard library.** Due to security considerations, some functions in the C standard library (*libc*) may have limited or restricted functionality, *e.g.*, perform system calls. As a result, the SGX SDK's tRTS comprises a modified version of libc that is specifically designed for use within SGX enclaves. This *trusted libc* (*tlibc*) provides a subset of libc functions that are considered safe to use within an enclave and essential for enclave development.

**Enclave building and debugging tools.** SGX enclave software can be compiled with a regular C/C++ compiler. The trusted part of an SGX application is built as a shared object (on Linux platforms), or a dynamic link library (on Windows platforms). The SGX SDK provides a signing tool which is used by the enclave author to sign the enclave using the author's public key. The enclave's signature is verified once the enclave is loaded, as well as during remote attestation as described before. The SGX SDK equally provides a tool for debugging enclave code during the development stage, as well as SGX simulation libraries which can be used for testing enclave code functionality without requiring physical SGX hardware support.

### *2.1.3 Arm TrustZone*

TEE technologies like Intel SGX, AMD SEV, and Intel TDX are tailored for server-end systems. In contrast, for edge-based systems like mobile devices and microcontrollers, *Arm TrustZone (TZ)* [159] is the predominant TEE technology.

TrustZone consists of hardware security extensions in ARM-based processors which divide the processor into two protection domains: *secure world* wherein sensitive operations can be performed, and *normal world* for performing non-sensitive operations, as illustrated in Figure 2.7. At any point in time, the processor operates exclusively in one of these worlds. A special bit known as the *non-secure (NS)* bit stored in the secure configuration register (SCR) determines the current protection domain of the processor, and is used for memory access control checks across both worlds. The processor can equally transition between worlds; TrustZone introduces a new component known as the *secure monitor* (operating in *monitor mode*) which acts as a bridge between both worlds and is responsible for storing processor state during transitions. A new privileged instruction, *i.e.*, *secure monitor call (SMC)*, allows software in both worlds to switch to the opposite world via monitor mode. Regarding interrupts, IRQ (normal interrupt request) and FIQ (fast interrupt request) in the secure world can also trigger a transition to monitor mode without an SMC. The ARMv8 architecture provides four privilege levels, *i.e.*, *exception levels* (EL) [102] at which code can run: EL0 for user space code, EL1 kernel space code, *e.g.*, the OS, EL2 for the hypervisor, and EL3 for secure monitor mode[3].

The memory infrastructure in TrustZone-enabled systems (Cortex-A) introduces a hardware component called the *TrustZone address space controller* (TZASC), which the Arm Trusted Firmware (TF-A) [103] uses to configure specific DRAM areas as secure regions. These configurations can be done such that secure world applications can access all memory regions while normal world applications are confined to non-secure memory. A similar memory partitioning functionality is performed by a component called *TrustZone memory adapter* (TZMA) but targets SRAM rather than DRAM. Both TZASC and TZMA may or may not exist on a specific system-on-chip (SoC) implementation. For example, the Raspberry Pi 3 plaftform supports some Arm TrustZone features but lacks TZASC and TZMA [181], and hence it lacks the capability of securing memory with TrustZone.

The security properties of TrustZone are particularly valuable in IoT systems where large amounts of security- and privacy-sensitive data are generated. More specifically, the memory partitioning functionality provided by TZASC can be employed to restrict peripheral memory in the secure world, thereby shielding sensitive data from unauthorised access.

## 2.2 GRAALVM

In the past, developers often faced limitations when it came to building and deploying programs, as they were constrained to using runtime environments specific to a particular programming language. For example the Java virtual machine (JVM) was primarily designed for efficient execution of Java programs, making it difficult to seamlessly integrate programs or software libraries written in other languages like Python, C, or C++ into the same JVM environment. This language-specific development approach also poses a challenge in creating generic tools, such as debuggers and program analysis tools, that could work across diverse runtime environments.

GraalVM is a novel tool that aims to remove the isolation between programming languages and enable interoperability in a shared runtime. It is a high-performance virtual machine (VM) developed by Oracle Labs, capable of running programs written in a wide range of programming languages. GraalVM provides a polyglot architecture that allows to transparently mix and match supported languages. For example, using GraalVM, one can easily invoke a JavaScript API from within a Java application and vice versa. This feature can be very useful when dealing with large-scale applications where different languages are better suited to different parts of the application, or when integrating legacy code into a new project.

At the heart of GraalVM is the *Graal compiler*, a dynamic just-in-time (JIT) compiler, written in Java, and capable of transforming bytecode into highly optimised machine code. Graal leverages techniques like partial evaluation, code inlining, and speculative optimisations to generate efficient machine code [199]. The Java Standard Edition (SE) version 9 platform introduced the Java virtual machine compiler interface (JVMCI) which allows custom compilers like Graal to be implemented as plugins into the regular Java HotSpot VM [91]. This allows the Graal compiler to be used by the HotSpot JVM as a replacement for the standard HotSpot JIT compilers. Figure 2.8 provides a summary of the key components that comprise the GraalVM architecture.

3 A prefix of "S" is usually added to the exception level for more precision when the code is being executed in the secure world. For example S-EL0/S-EL1 explicitly indicate that the user space/kernel space code is executing in the secure world.

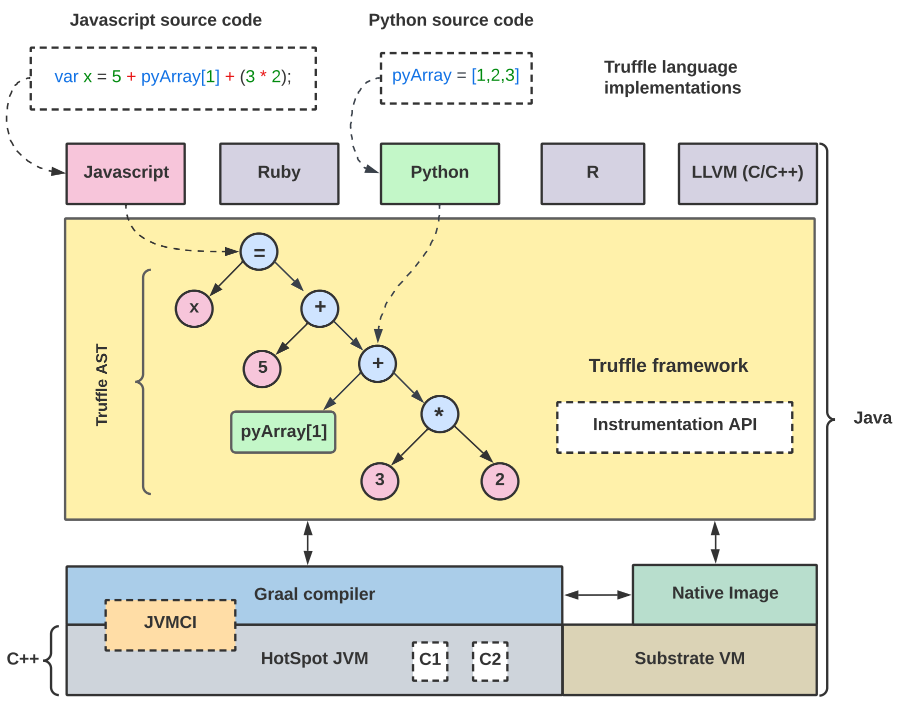


**Figure 2.8:** GraalVM architecture.

### *2.2.1 Truffle framework*

When considering the multi-language capabilities of GraalVM, a common question that emerges is what methodology it employs. At the foundation of GraalVM's multi-language capabilities is the Truffle language implementation framework, *i.e.*, TRUFFLE. The latter is an open source Java library for building tools and programming language implementations as interpreters for *abstract syntax trees*. An abstract syntax tree (AST) is a tree-like representation of the syntax of a program, where each tree node corresponds to a syntactic element of the programming language, such as a function call, a loop, a conditional statement, a variable assignment, *etc*. Truffle framework provides common AST nodes which form the building blocks of all languages supported by GraalVM's runtime. These languages are referred to as TRUFFLE languages, and examples include Java, JavaScript, Python, Ruby, R, as well as LLVM based languages like C and C++. Thus, programs written in all TRUFFLE languages are transformed into a common AST representation called a *Truffle AST*, which is processed and interpreted/compiled into efficient machine code in a language-agnostic manner by the Graal compiler. Truffle framework also provides an API to instrument nodes in a Truffle AST at runtime. The common AST representation together with this instrumentation API can be leveraged to build language-agnostic tools like debuggers [137], profilers [137], or programming analysis tools [92], as we will see subsequently.

### *2.2.2 Native images*

GraalVM provides a tool called *native-image* which allows to ahead-of-time (AOT) compile Java programs into standalone executables called *native images*. GRAALVM native-image uses a static analysis approach called *points-to analysis* [211] to find the reachable program elements, *i.e.*, classes, methods, and fields, and AOT compiles only these reachable elements into the final native image. In addition, GRAALVM native-image makes it possible to run class initialisation code at build time (*i.e.*, native image build time), instead of at runtime, and makes available all pre-initialised application objects to the final native image at runtime. The overall result of this approach is reduced runtime memory footprints and startup times for native images, relative to regular Java applications running in the JVM.

GraalVM native-image makes a closed-world assumption, *i.e.*, it considers that all application classes that can be executed at run time are known and available at build time. To support dynamic features such as reflection, the user provides a list of the classes, fields, and methods that can be accessed dynamically. Each element of this list is then always included in the native image, in addition to all classes, fields and methods transitively reachable from these elements. This list can be provided through CLI options, programmatically, or a JSON file. GraalVM native-image provides a *tracing agent* [152] which assists developers in generating such a JSON file.

GraalVM native images do not run on a regular JVM (*e.g.*, HotSpot): runtime components that are needed to run JVM-based applications, such as a garbage collector (GC), support for thread scheduling and synchronisation, as well as stack walking and exception handling are directly included inside the created native images. These minimal runtime components are collectively referred to as *Substrate VM* [134]. From a security point of view, including only reachable components in the binary can be beneficial when the intended runtime is a TEE. This makes native images suitable for an enclave runtime, as will be discussed in the subsequent chapter.

### *2.2.3 Adding TEE capabilities for high level programming languages*

As mentioned earlier in this chapter, a major challenge with TEE related development is the low level nature of development tools (*i.e.*, based on C/C++). However, we can leverage the flexibility offered by GraalVM to break this language barrier preventing TEE adoption for a wider range of programming languages. For example, the AOT capability of GraalVM can be leveraged to secure a Java based program inside an Intel SGX enclave (see Chapter 3), while Truffle can be used to develop multi-language tools to analyse and partition programs to be run inside an enclave (see Chapter 4).

## 2.3 NON-VOLATILE MEMORY

While traditional DRAM provides fast (latencies of $\approx 80-100ns$[169]) and fine-granular access to memory for applications, it is *volatile*; meaning all data on it is lost upon a power interruption. On the contrary, *secondary storage*, *e.g.*, solid state drive (SSD) or hard disk drive (HDD) provides *persistence* (*i.e.*, retains data even without power) and offers higher storage capacity at a relatively lower cost compared to DRAM. However, this comes at the cost of slower access speeds, *e.g.*, $\approx 10-100us$ in NAND SSD[169]. Over the years, research endeavors have been dedicated to finding innovative solutions that can combine the performance benefits of volatile memory with the data persistence of secondary storage.

*Non-volatile memory* (NVM) is a novel memory technology that is *non-volatile* and *byte-addressable*. *Non-volatile* means data written can survive a power failure or system crash, and *byte-addressable* means data can be accessed at byte-granularity. To better appreciate byte-addressability, consider an application that modifies $64B$ of data which must be saved to block storage, *e.g.*, a SSD. Because storage based accesses only happen using block I/O (input/output), usually $4KB$ block sizes, writing the $64B$ of data to storage requires writing the entire $4KB$ containing-block to storage. Similarly, reading $64B$ of data from storage will require reading the entire $4KB$ block containing the $64B$ of data. This is not efficient from a performance perspective. The byte-addressability of NVM makes it possible to write/read exactly $64B$ of data to/from NVM, instead of the entire $4KB$ block. NVM is also commonly referred to as *persistent memory* (PM). Going forward, we will use both terms interchangeably.

As byte-addressable memory, PM can be accessed directly using CPU `load` and `store` instructions, just like DRAM. As such, applications can write/read data to/from PM without relying on expensive system calls, *e.g.*, `read`, `write`, *etc*.

PM sits between DRAM and SSD in terms of performance and cost parameters [227]. Although its write latency is greater than that of DRAM, the cost per bit is lower. From a storage standpoint, PM is faster but comes with a higher price tag compared to SSD.

Several PM technologies exist: 3D XPoint [218], phase change memory (PCM), resistive RAM (ReRAM), and spin-torque transfer RAM (STT-RAM). The most popular commercially available PM implementation is *Intel Optane DC Persistent Memory* [73] (based on 3D XPoint), released in April 2019.

### *2.3.1 Leveraging PM in TEE based applications*

A major source of overhead in TEEs like Intel SGX is enclave context switches to perform system calls, for example to read data from, or write data to secondary storage. Leveraging the byte-addressability of PM removes the need for expensive context switches to access storage in an Intel SGX application. This idea is leveraged in Chapter 6 to improve the performance of enclave-based applications.

# Part I

## TEE PROGRAM PARTITIONING

CHAPTER 3

# Partitioning Java Applications for Enclaves

This chapter presents MONTSALVAT, a tool to automatically partition Java programs for SGX enclaves. MONTSALVAT leverages Java annotations to specify security sensitive classes, and bytecode transformations to partition the program. The partitioned components are AOT compiled into binaries using GraalVM, and used to build the final Intel SGX enclave program. The evaluation of MONTSALVAT demonstrates its usefulness with real-world programs.

This chapter is organised as follows:

## 3.1 INTRODUCTION

The Java programming language is widely used in cloud infrastructures, with popular cloud frameworks such as Apache Hadoop [44], Apache ZooKeeper [46], Apache Spark [45], and more based on Java. As several cloud-based data processing applications increasingly rely on these frameworks, ensuring the security and privacy of the applications represents an important challenge. While the introduction of TEEs like Intel SGX has the potential to mitigate these security concerns, the limited availability (or lack thereof) of Java based TEE development tools poses a significant roadblock for developers.

As previously detailed in Chapter 1 (§1.1), the conventional approach to address this problem involves running entire Java applications, including the JVM, inside the enclave by relying on a library OS like Gramine [194], SGX-LKL [160], Haven [11], or tools such as SCONE [7]. This approach offers good compatibility for legacy applications and demands minimal developer intervention, but it has the inherent drawback of significantly increasing the TCB [177]. This violates the principle of least privilege [167] and increases the likelihood of enclave vulnerabilities. Ideally, the application should be *partitioned* into trusted and untrusted components which run in and out of the enclave respectively. This can be done either *manually* or *automatically*.

On the one hand, contemporary manual partitioning tools [30, 130, 54, 85] are primarily tailored for lower-level programming languages like C and C++. On the other hand, even when such manual partitioning tools are available, dealing with a managed language like Java is very challenging for the following reasons:

1. Code running outside of an enclave (in the untrusted runtime) may allocate objects inside the enclave (the trusted runtime), and code running inside the enclave may allocate objects outside of the enclave. Since both runtimes operate on separate memory heaps, there is a need for an efficient mechanism to ensure object communication across the two runtimes.
2. Since there may be references between the untrusted runtime and the enclave, the garbage collector has to be extended to ensure consistency, *i.e.*, objects in one runtime should not be destroyed if objects in the opposite runtime still reference them.

These challenges, coupled with the extensive reachability of Java classes and the complexities of the JVM, highlight the necessity for automatic or semi-automatic tools for partitioning Java code for enclaves. While a few tools [197, 79] have been proposed to tackle code partitioning in Java, they still suffer from large TCBs or do not adequately address the aforementioned challenges.

In this work, we introduce MONTSALVAT, a tool designed to automatically partition Java applications for Intel SGX enclaves.

MONTSALVAT leverages Java code annotations and bytecode transformations to split Java applications into trusted and untrusted components, and applies distributed techniques like remote method invocation [58] to enable efficient communication between trusted and untrusted objects. MONTSALVAT leverages GraalVM's *native-image* tool to ahead-of-time (AOT) compile applications into *native images* [42, 138, 211], which do not require a full-blown JVM at runtime.

GRAALVM native-image excludes unused classes, methods, and fields from the application binary, thus reducing the application's attack surface. To ensure consistent garbage collection across the trusted and untrusted runtimes, MONTSALVAT employs *garbage collection helper threads*. These threads periodically check for application objects that have been garbage-collected or are eligible for collection in one runtime, and trigger the necessary enclave transitions to inform the garbage collector in the opposite runtime. Our evaluation of MONTSALVAT with micro-benchmarks, and Java frameworks like PalDB [105] and GraphChi [93] shows real-world Java applications can be successfully partitioned without compromosing performance.

## 3.2 BACKGROUND

### *3.2.1 GraalVM Native Image*

As previously covered in Chapter 2 (§2.2.2), GraalVM Native Image (or GraalVM native-image) is a tool, built on top of the GraalVM compiler [13], to compile ahead-of-time applications into standalone executables, which are named *native images*. It supports JVM-based languages, *e.g.*, Java, Scala, Clojure and Kotlin. Native images leverage static analysis to include only reachable application classes, methods, and fields in the final binary. GraalVM native-image enables applications to execute initialisation code (*e.g.*, reading and parsing a configuration file) at build time, effectively reducing the application startup as less logic is executed at runtime. To transfer the result of the initialisation (Java objects) from build to runtime, GraalVM native-image takes a snapshot of the heap (called the *image heap*) at the end of the build, and stores it into the generated executable. The image heap is memory mapped inside the application heap at startup, allowing the application to start from the state initialised at build time.

**GraalVM Isolates.** GraalVM native images do not run on a regular JVM (*e.g.*, HotSpot): runtime components that are needed to run JVM-based applications, such as a garbage collector, support for thread scheduling and synchronisation, as well as stack walking and exception handling are directly included inside the created native images. GraalVM native-image provides the possibility of creating multiple independent VM instances at runtime, which are called *isolates*. Each isolate operates on a separate heap, allowing garbage collection to be performed independently. Thus, threads executing in one isolate are not affected by garbage collection done in another isolate. GraalVM native-image also provides the `@CEntryPoint` annotation to specify *entry point* methods [141] that should be exported and callable from a C API, thus enhancing interoperability between lower-level languages. Montsalvat creates a default isolate for each of the two components of the partitioned Java application (trusted and untrusted), which provides the execution contexts for all entry point methods, *e.g.*, `main`.

## 3.3 THREAT MODEL

Montsalvat assumes enclave code and the CPU package are trusted, similar to related work with SGX [7, 104, 197, 79, 170]. The source code annotation, image build via AOT compilation, and final enclave signing are done in a trusted environment. This prevents malicious classes/bytecode from being introduced into the enclave at runtime [121]. The integrity of the enclave can then be validated at runtime via remote attestation [20, 34] mechanisms.

Montsalvat supports a powerful adversary with control over the full software stack, including the OS, hypervisors, and access to the physical hardware, *e.g.*, DRAM, secondary storage, *etc*. The adversary's goal is to gain access to confidential data (*e.g.*, passwords, encryption keys, *etc.*) which may be processed in trusted application classes, or to damage the integrity of confidential data.

Montsalvat is resilient to physical attacks like cold boot attacks [64] aimed at reading sensitive data in DRAM, or bus probing [217] to read the memory channel between the CPU and DRAM; the SGX security model [34] prevents these.

We assume the adversary cannot physically open and manipulate the SGX-enabled processor package (as in [21]), and that the enclave code does not intentionally leak sensitive data. Denial-of-service

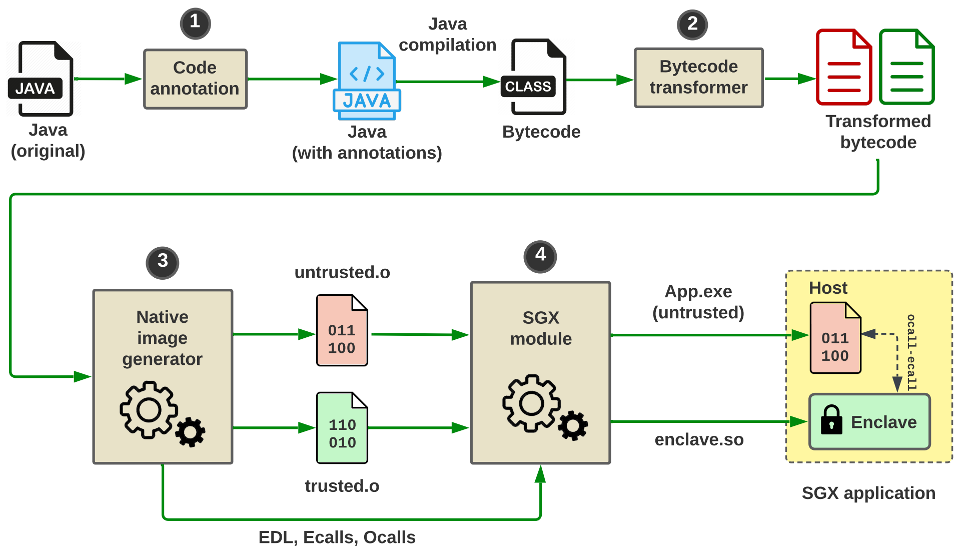


**Figure 3.1:** Overview of MONTSALVAT's workflow.

and side-channel attacks [16, 171], for which mitigations exist [128, 60], are considered out of scope.

## 3.4 ARCHITECTURE

The main goal of MONTSALVAT is to partition Java applications for SGX enclaves. The final partitioned application comprises a *trusted* and an *untrusted* part, respectively running inside and outside the enclave. Figure 3.1 depicts MONTSALVAT's complete workflow, from the source code to the generation of the final SGX application. It comprises 4 main phases: (❶) code annotation (see §3.4.1), (❷) bytecode transformation (see §3.4.2), (❸) native image partitioning (see §3.4.3), and (❹) SGX application creation (see §3.4.4).

To illustrate the inner workings of MONTSALVAT, we consider a synthetic Java application to be partitioned (see Listing 1). Three classes mutually interact via method calls: classes `Account` and `AccountRegistry` perform sensitive operations, and thus need to be secured in the enclave. Conversely, class `Person` is untrusted and will not be included in the enclave.

### *3.4.1 Code annotation*

When partitioning applications destined for enclaves, an important question to answer is how to specify what should be secured or not. We propose a technique based on *class annotations*. Classes serve as a fundamental building block for object-oriented applications, and it is very intuitive to reason about security along class boundaries. Through annotations, developers can easily specify which classes need to be secured and which ones do not.

MONTSALVAT supports two principal annotations: `@Trusted` and `@Untrusted`,[1] which developers can use to specify secure and insecure classes, respectively. In Listing 1, `Account` and `AccountRegistry` classes are annotated as trusted, whereas class `Person` is untrusted.

A trusted class is always instantiated and manipulated inside the enclave, which has two main implications. First, its member fields which are not instances of untrusted classes are allocated on the enclave heap. Second, its methods are always executed inside the enclave.

1 Montsalvat annotations

```
 1  @Trusted
 2  public class Account {
 3    private String owner;
 4    private int balance;
 5    public Account(String s, int b) {
 6      this.owner = s;
 7      this.balance = b;
 8    }
 9    public void updateBalance(int v) {
10      this.balance += v;
11    }
12  }
13
```

```
14  @Trusted
15  public class AccountRegistry {
16    // Trusted obj in trusted obj
17    private List<Account> reg =
18    new ArrayList<Account>();
19    public AccountRegistry() {}
20    public void addAccount(Account a) {
21      this.reg.add(a);
22    }
23  }
24
```

```
25  @Untrusted
26  public class Person {
27    private String name;
28    // Trusted obj in untrusted obj
29    private Account account;
30    public Person(String s, int v) {
31      this.name = s;
32      this.account = new Account(s, v);
33    }
34    public Account getAccount() {
35      return this.account;
36    }
37    public void transfer(Person p, int v) {
38      p.getAccount().updateBalance(v);
39      this.account.updateBalance(-v);
40    }
41  }
42
```

```
43  @Untrusted
44  public class Main {
45    public static void main(String[] args) {
46      Person p1 = new Person("Alice", 100);
47      Person p2 = new Person("Bob", 25);
48      p1.transfer(p2, 25);
49      AccountRegistry reg =
50      new AccountRegistry();
51      reg.addAccount(p1.getAccount());
52    }
53  }
54
```

**Listing 1:** Illustrative example with annotated classes using MONTSALVAT: trusted classes (lines 1-24), and untrusted (lines 25-54).

Similarly, an untrusted class has its instance objects allocated only on the untrusted heap, along with all its member fields which are not instances of trusted classes. All its methods are executed outside the enclave.

MONTSALVAT maintains a single version of a trusted or an untrusted object in both worlds by leveraging *proxy* objects (see §3.4.2). In our programming model, some classes can be *neutral* as they may not be inherently trusted or untrusted. This is the case for utility classes like `Array`, `Vector`, `String` or other similar application-specific classes added by the developer.

Such classes are not inherently security-sensitive and can be accessed in or out of the enclave without the use of proxies. Contrary to trusted and untrusted classes, neutral class instances can have several copies in both worlds and may evolve independently. The `@Neutral` annotation is optional, *i.e.*, classes that are not annotated are by default neutral.

One may legitimately question the relevance of two annotations, thinking the `Trusted` annotation is sufficiently expressive. Our argument for an `@Untrusted` annotation is twofold: *(1)* some classes may perform many system-related operations that are not supported inside enclaves, and keeping them in the enclave needlessly increases the TCB as they will perform many `ocall` transitions to the outside; *(2)* classes which could introduce potential security vulnerabilities in the enclave should preferably be kept out of the enclave. The `@Untrusted` annotation solves these problems, while also allowing for easy distinction with neutral classes.

**Assumptions.** We assume all *annotated* classes are properly encapsulated, *i.e.*, class fields are private. On the one hand, this prevents complex and expensive data flow analysis to ensure sensitive class fields do not leave the enclave. On the other hand, it guarantees that all class fields can only be accessed from outside classes via public getters and setters exposed by the class. As such, it is easier to control

```java
public class Person {
  private int hash;
  public Person(String s, int v) {
    byte[] buf = serialise(s);
    CCharPointer ptr = getPointer(buf);
    this.hash = getHash(this);
    ocall_relayPerson(this.hash, ptr, v);
  }
  public void Account getAccount() {
    ocall_relayGetAccount();
  }
  public void transferPerson(Person p, int v) {
    ocall_relayTransferPerson(p.getHash(), v);
  }
}
```

**Listing 2:** Proxy for the untrusted class `Person`.

```java
public class Account {
  private int hash;
  public Account(String s, int b) {
    byte[] buf = serialise(s);
    CCharPointer ptr = getPointer(buf);
    this.hash = getHash(this);
    ecall_relayAccount(this.hash, ptr, b);
  }
  public void updateBalance(int v) {
    ecall_relayUpdateBalance(this.hash, v);
  }
}
```

**Listing 3:** Proxy for the trusted class `Account`.

access to these sensitive class fields by applying techniques such as transparent encryption/decryption at the level of these public methods. Given that encapsulation is a fundamental aspect of object orientation, we find this assumption to be reasonable.

### *3.4.2 Bytecode transformation*

The artefacts of the partitioned application consist of two native images: a *trusted* and an *untrusted* image. The trusted image does not have any untrusted functionality, and the untrusted image does not have any trusted functionality. However, trusted objects (*i.e.*, instances of trusted classes) may call untrusted objects (*i.e.*, instances of untrusted classes) and vice versa. Hence, we need to have a bidirectional communication mechanism for code flow execution. For that purpose, we introduce the notion of *proxy classes*: instances of untrusted classes have proxies in the trusted runtime, and conversely instances of trusted classes have proxies in the untrusted runtime. These proxies serve as gateways to access the functionalities (*i.e.*, methods) of their real class in the opposite runtime.

The proxy classes expose the same methods as the original classes and replace the method implementations by a transition logic to access the original functionalities across enclave boundaries. This design makes cross-enclave object communication easier and helps maintain the object-oriented nature of the program as a whole after it is partitioned. We rely on bytecode transformations to *create* these proxy classes and *inject* code into existing classes to implement the enclave transitions. MONTSALVAT uses Javassist [77], a popular bytecode transformation framework, to achieve this phase.

MONTSALVAT introduces matching proxy classes for all trusted and untrusted classes. The points-to analysis of GRAALVM native-image automatically prunes/removes proxies for classes that are not reachable, thus unnecessary proxy classes are removed. As GRAALVM does not include unreachable proxy classes in the generated native images (see §3.4.3), we did not include that analysis in the bytecode transformer. Listings 2, 3 and 4 illustrate the result of bytecode transformations for the corresponding classes.

```java
public class Account {
  private String owner;
  private int balance;
  public Account(String s, int b) {...}
  public void updateBalance(int v) {...}

  @CEntryPoint
  public static void relayAccount(Isolate ctx, int hash, CCharPointer buf, int b) {
    String s = deserialise(buf);
    Account mirror = new Account(s, b);
    mirrorProxyRegistry.add(hash, mirror);
  }
  @CEntryPoint
  public static void relayUpdateBalance(Isolate ctx, int hash, int v) {
    Account mirror = mirrorProxyRegistry.get(hash);
    mirror.updateBalance(v);
  }
}
```

**Listing 4:** Concrete class `Account`, once transformed.

For the purposes of the trusted image, this process creates proxy classes for untrusted classes by stripping the methods (*i.e.*, removing the method bodies) of the untrusted classes. Listing 2 shows the corresponding proxy class for the untrusted class `Person` in our illustrative example. The bodies of the stripped methods are replaced with native routines which perform `ocall` transitions to the corresponding method in the untrusted runtime (lines 7, 10, 13 in Listing 2). Analogously, for untrusted image generation, the bytecode transformer creates proxy classes for trusted classes by stripping all methods of trusted classes, and replacing the method bodies with native methods which perform `ecall` transitions. Listing 3 shows the corresponding proxy class for the trusted class `Account` in our illustrative example. The proxy class fields are removed and a `hash` field is added to each proxy class which stores the hash of the proxy object (*i.e.*, line 6 in Listing 3). Our present implementation uses a hash function based on Java identity hash codes. To minimise hash collisions, a hashing algorithm like MD5 [165] should be used. The result of the stripping operations is the removal of all untrusted functionality from the trusted partition, and all trusted functionality from the untrusted partition. Only annotated classes are modified by the bytecode transformer, *i.e.*, neutral classes are not changed.

From this point onward, we refer to all unstripped classes as *concrete classes* and stripped classes as *proxy classes*. We respectively call the instances of these classes *concrete objects* and *proxy objects*. If a concrete object in one runtime (trusted or untrusted) has a correspondence with a proxy object in the opposite runtime, we refer to that concrete object as a *mirror object*, *i.e.*, the proxy's mirror copy.

**Relay methods.** For the methods in one runtime (a native image) to be callable from the other runtime (another native image), these methods must be exported as entry points. These entry point methods must be static, they may only have non-object parameters and return types, *i.e.*, primitive types or `Word` types (including pointers) [138], and they must specify the GraalVM isolate that serves as execution context for the method. As a result of these restrictions, it is not feasible to export all methods of concrete classes as entry points directly, since this would require changing their signatures. To circumvent this limitation, we introduce static entry point methods to act as wrappers for the invocations of the associated class or instance methods. We call these *relay methods*.

For every public method in a concrete class, including all constructors, the bytecode transformer adds an associated relay method to the class. The native methods (*i.e.*, `ecall` and `ocall` routines) added to the stripped methods of the proxy classes perform enclave transitions to invoke the corresponding relay methods.

The parameters of a relay method comprise: an isolate which provides the execution context for the method call, the hash of the calling proxy object (for non-static methods), all primitive parameters of

```java
public void addAccount(Account acc) {
  ecall_relayAddAccount(acc.getHash());
}

@CEntryPoint
public static void relayAddAccount(Isolate ctx, int hash) {
  Account mirror = mirrorProxyRegistry.get(hash);
  this.addAccount(mirror);
}
```

**Listing 5:** Proxy (top) and relay (bottom) methods for the `AccountRegistry` class.

the associated method, and pointers (*i.e.*, `CCharPointer` [140]) which represent the addresses of buffers obtained from the serialisation of any object parameters which are instances of neutral classes (as these classes do not need proxies). For proxy object parameters, the hash of the corresponding proxy is passed as parameter and the corresponding mirror object is used as the parameter once the real (*i.e.*, concrete) method is called in the opposite runtime. Similarly, for mirror object parameters, the hash of the corresponding proxy is also sent, and the proxy object is used in the opposite runtime as parameter in the method.

As for the serialised buffers, they are deserialised and the corresponding object parameter recreated in the body of the relay method. For relay methods of constructors, we instantiate the corresponding mirror object, and add the mirror object's strong reference and associated proxy hash to a global registry, which we call the *mirror-proxy registry*. For instance methods, we add code to look up the corresponding mirror object in the registry, and then invoke the instance method on that mirror object with its corresponding parameters. Neutral object return types from the untrusted runtime are also serialised and copied across the enclave boundary. Both the trusted and untrusted runtimes have a mirror-proxy registry.

The code in Listing 4 outlines the state of concrete class `Account` after bytecode transformation. For illustration, the relay method `relayAccount` (line 8) is added into concrete class `Account` in the trusted runtime automatically.

We complete the transformation of the other classes of our illustrative example as follows. For proxy class `Person`, we have `ocalls` instead to the relay methods. For proxy class `AccountRegistry`, we have a proxy `Account` as parameter in the `addAccount` method, and only its hash is passed to the opposite runtime. The resulting proxy method and the corresponding relay method in concrete class `AccountRegistry` are shown in Listing 5.

**Native transition methods.** The native transition methods (*e.g.*, `ecall_addAccount`) are C routines which perform enclave transitions to the opposite runtime. During bytecode transformation, the definitions of the native transition methods are absent and only their signatures are provided. Their definitions are generated by the native image generator, as will be seen subsequently.

### *3.4.3 Native image partitioning*

The GraalVM native image generator is responsible for building native images. It takes as input compiled application classes (bytecode) and all their associated external libraries, including the JDK. The native image generator then performs static analysis to find the reachable program elements, *i.e.*, classes, methods, and fields. Only the reachable elements are then compiled ahead-of-time into the final native image. This is advantageous for TEE-based application since it excludes any redundant application logic from the enclave. The resulting native image embeds runtime components for garbage collection (memory management), thread scheduling, *etc*.

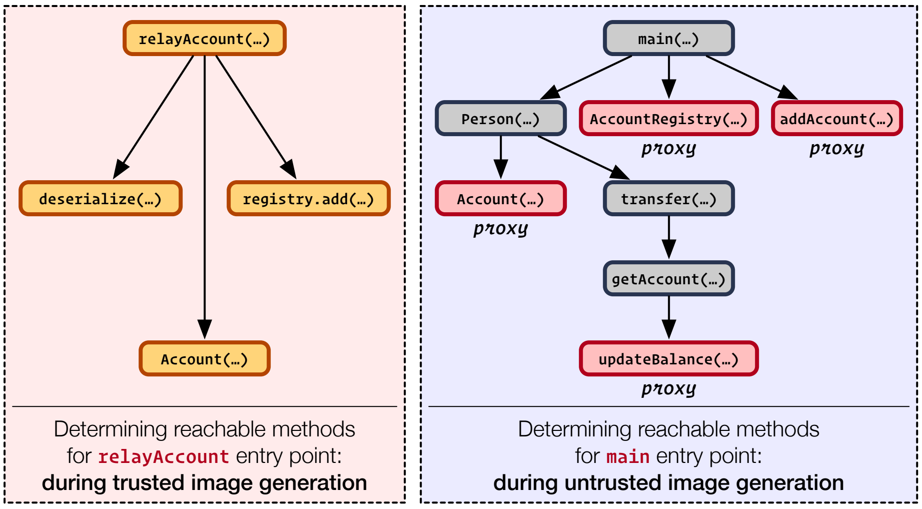


**Figure 3.2:** Simplified reachability analysis done for two entry point methods (`relayAccount` and `main`).

**Static analysis for trusted and untrusted image generation.** The bytecode transformations produced two sets of class files: the first set $(T)$ comprises modified trusted classes and untrusted proxy classes, while the second set $(U)$ comprises modified untrusted classes and trusted proxy classes. The unannotated/neutral classes $(N)$ were not changed by the bytecode transformer. These three sets of classes are used by GraalVM to generate two native images that collectively represent the partitioned application.

The native image generator in Montsalvat uses set $(T \cup N)$ as input for *trusted image* generation and the set $(U \cup N)$ as input for *untrusted image* generation. To determine reachable program elements (*i.e.*, classes, fields and methods) the native image generator leverages a static analysis technique known as points-to analysis [211, 178]. Points-to analysis starts with all entry points and iteratively processes all transitively reachable classes, fields, and methods [211]. For the sake of brevity, we do not include all the steps performed during points-to analysis in GraalVM native-image (see [211] for details). For the trusted image, all the relay methods of trusted classes serve as entry points (recall the `@CEntryPoint` annotation, *e.g.*, in Listing 4). For the untrusted image, the main entry point (Java application's `main` method) and the relay methods of untrusted classes serve as entry points. Conceptually, we can include the main entry point in either the trusted or untrusted image. However, we chose to add it in the untrusted image because: *(1)* it prevents an `ecall` transition to invoke the main method and an `ocall` transition to create garbage collection helper threads (see §3.4.5) once in the main method, and *(2)* it is in accordance with the Intel SGX programming convention where applications begin in the untrusted runtime.

Figure 3.2 illustrates a simplified reachability analysis done for two entry point methods (`relayAccount` and `main`) to determine other reachable methods. The graph shown is a subset of the full graph; other methods are made reachable at the leaf nodes. For the trusted image, the relay methods (which are entry points) in the trusted classes ensure that other trusted class methods, as well as methods from neutral classes (*e.g.*, `serialise`, `registry.add`, *etc.*), are reachable. Similarly, for the untrusted image, the main entry point ensures that the `Person` class methods, as well as methods from proxy classes (`Account` and `AccountRegistry`), are reachable. A similar process is performed for all other entry points.

Once static analysis is complete, the trusted image no longer contains untrusted methods/functionality. It embeds proxies instead, in case some untrusted proxy class methods were reachable. Similarly, the untrusted image does not contain trusted methods, but only proxies to those if some proxy class methods were reachable. Following from our illustrative example, proxy class `Person` is not included

```
void ecall_relayAddAccount(int hash) {
  Isolate ctx = getIsolate();   // Get the enclave isolate
  relayAddAccount(ctx, hash);
}
```

**Listing 6:** C code for `ecall_relayAddAccount`.

inside the trusted image since it is not reachable from any of the trusted classes.

In the `main` method in our illustrative example, upon object creation at runtime, the constructors of `Person` $p1$ and $p2$ instantiate a proxy object of the *proxy* `Account` class. The string parameters ‘‘`Alice`’’ and ‘‘`Bob`’’ are serialised and an `ecall` transition is done to create a corresponding *mirror* `Account` object in the enclave. Similarly, when `p1.transfer(p2, 25)` is invoked, the corresponding proxy `Account` objects perform enclave transitions to update the balances in the enclave. In the same way, the call to the proxy `AccountRegistry` constructor performs a transition to create a mirror object in the enclave corresponding to proxy object `reg`. The latter performs a transition too when `addAccount` is called.

By default, the native image generator compiles all reachable methods and links the latter with GRAALVM's native libraries to produce an executable or a shared object file. We modified the image generator to bypass the linking phase that produces executables or shared objects, so as to produce relocatable object files (`.o`) which can be linked to other libraries to build the final SGX application. The resulting images for the trusted and untrusted parts are `trusted.o` and `untrusted.o` respectively. These are dispatched to the SGX module and used to build the final SGX application.

**SGX code generator.** During bytecode transformation, native `ecall` and `ocall` transition routines are added to proxy classes. We extended GRAALVM native-image with a class to generate C code definitions for the corresponding `ecall` (added in trusted proxy classes) or `ocall` (added in untrusted proxy classes) transitions, as well as their corresponding header files. Listing 6 shows the generated C code for the
`ecall_relayAddAccount` method.

The code generator also creates associated EDL files used by the *Edger8r* tool in the Intel SGX SDK to build edge routines, which marshal data (*e.g.*, serialised method parameters) across the enclave boundary.

### *3.4.4 SGX application creation*

This is the final stage MONTSALVAT's workflow. Because SGX enclaves operate only in user mode, they cannot issue system calls and standard OS abstractions (*e.g.*, file systems, network), which are ubiquitous in real-world applications. The solution to this problem is to relay these unsupported calls to the untrusted runtime, which does not have the same limitations. In contrast to systems that introduce a *LibOS* (*i.e.*, an entire operating system implemented as a library) in the enclave, we leverage an approach which involves redefining unsupported `libc` routines as wrappers for `ocalls`. These redefined `libc` routines in the enclave constitute MONTSALVAT's *shim library*.[2] The latter intercepts calls to unsupported `libc` routines and relays them to the untrusted runtime. A *shim helper library* in the untrusted runtime then invokes the real `libc` routines. This by design reduces the TCB when compared to *LibOS*-based systems.

MONTSALVAT then compiles all generated `ecall` routines and statically links them with the trusted image (`trusted.o`), the shim library and native libraries from GRAALVM to produce the final enclave

2 Montsalvat shim library

shared library, which corresponds to the trusted part of the Java SGX application. Similarly, the generated `ocall` routines are also compiled and linked with the untrusted image (`untrusted.o`) and GraalVM native libraries to produce the final untrusted component. In accordance with Intel SGX's application model, Montsalvat compiles the main entry point of partitioned applications in the untrusted component. The resulting trusted and untrusted components compose the final SGX application. At runtime, a GraalVM isolate is created in both the trusted and untrusted part of the application. These isolates provide the execution contexts for transition routines, *i.e.*, the trusted isolate serves `ecall` routines while the untrusted isolate serves `ocall` routines.

### *3.4.5 Garbage collection*

Following our partitioned application design, untrusted code objects can have trusted counterparts (proxies) and vice versa. This presents a challenge at the level of garbage collection (GC) because we must ensure synchronised destruction of objects across the trusted and untrusted heaps. More specifically, we need to synchronise GC of proxy and mirror objects, *e.g.*, the mirror of proxy object `reg` should not be destroyed before `reg` in our illustrative example (Listing 1). Similarly, when `reg` is destroyed, its corresponding mirror object should be made eligible for GC.

The Java programming language specification provides *finalizer methods* [151] which the garbage collector can invoke prior to garbage collecting an object. So one could envision a solution based on finalizer methods. However, the latter are deprecated since Java 9 and have badly designed semantics [133]. For example, a finalizer method can make a proxy object reachable again, which may lead to an inconsistent state across the trusted and untrusted heaps after the proxy's mirror object is destroyed.

To address this problem, we implemented an application-level GC helper based on *weak references*. GC helper threads (one for each partition) are created on application initialisation and are used to track proxy and mirror object creation and destruction.

**Strong and weak references in Java.** When an object is created in Java, memory space is allocated for it in the heap. References in Java serve as pointers, *i.e.*, memory addresses that enable access to this allocated memory space. The Java programming language provides various types of references for objects, with *strong* and *weak* references being the most common. A strong reference is the default reference type in Java; existence of a non-null strong reference to a Java object prevents the garbage collector from reclaiming the memory occupied by the object. For example, `Person p = new Person(...);` creates a `Person` object with an associated strong reference `p`. This object is not made eligible for garbage collection unless `p` is set to `null`. Conversely, a weak reference in Java allows the associated object to be reclaimed by the garbage collector if there is no non-null strong reference to it. Weak references are useful in scenarios where we wish to track objects in a cache *e.g.*, a HashMap, while allowing the objects to be eligible for garbage collection. For example, `WeakReference<Person> weakRef = new WeakReference<>(p);` creates a weak reference to the object referenced by `p`.

**Garbage collection consistency in Montsalvat.** When a proxy object is created, Montsalvat stores a weak reference and the hash of the proxy object in a global list. We use a weak reference here because it does not prevent the proxy object from being garbage collected once it is eligible for garbage collection. The corresponding GC helper thread periodically (*e.g.*, every second) scans this list for null referents of weak references, *i.e.*, objects referred to by the weak references. Once such a null referent is found, it means the proxy object has been (or is eligible for being) garbage collected, and thus we can remove the corresponding mirror object from the mirror-proxy registry in the opposite runtime.[3]

3 Montsalvat GC helper

This makes the mirror object eligible for GC if it is not strongly referenced anywhere else. Both the trusted and untrusted runtimes maintain independent lists of proxy weak references for the associated runtime, and the two garbage collection helper threads spawned at application initialisation scan these lists throughout the application's lifetiime.

Algorithm 1 outlines the work of a GC helper thread.

**Algorithm 1** GC helper thread.

```
1:  $proxyWeakRefs \leftarrow \{list\ of\ proxy\ obj\ weak\ refs\}$
2:  while !stopped do
3:      $wait(TIME)$
4:      for all $ref \in proxyWeakRefs$ do
5:          if $ref.get() = null$ then
6:              $proxyId \leftarrow getProxyId()$
7:              if $inEnclave$ then
8:                  $ocall\_removeMirror(proxyId)$
9:              else
10:                 $ecall\_removeMirror(proxyId)$
11:             end if
12:             $proxyWeakRefs \leftarrow proxyWeakRefs \setminus ref$
13:         end if
14:     end for
15: end while
```

The `removeMirror` ocalls or ecalls perform an enclave transition which removes corresponding pairs of mirror object and proxy id from the mirror-proxy registry. This automatically makes the mirror object eligible for GC if it is not strongly referenced somewhere else in the program.

### *3.4.6 Running unpartitioned native images*

Despite the benefits of partitioning an enclave application, situations may arise where it is much easier for the application developer to run the entire application as a native image inside the enclave. This could happen when the majority of classes potentially deal with sensitive information and no classes qualify as untrusted. Consequently, MONTSALVAT makes it possible to run unpartitioned applications in the enclave. Neither code annotations nor bytecode modifications are required in such cases. The original application is built into a single native image which is linked to the final enclave object.

### *3.4.7 Prototype implementation*

Our current MONTSALVAT prototype is based on GRAALVM CE v21.0.0 for Java 8. GRAALVM and the bytecode transformer are implemented in Java. Our modifications in GRAALVM amount to $\approx 1,400$ lines of code (LoC). The bytecode transformer[4] relies on Javassist v3.26 and contains $\approx 2,100$ LoC. The SGX module.[5] is based on SGX SDK and SGX driver v2.11. It consists of $\approx 10,200$ C/C++ LoC. A prototype of MONTSALVAT has been released as open-source.[6]

4 Bytecode transformer
5 SGX module
6 Montsalvat repository

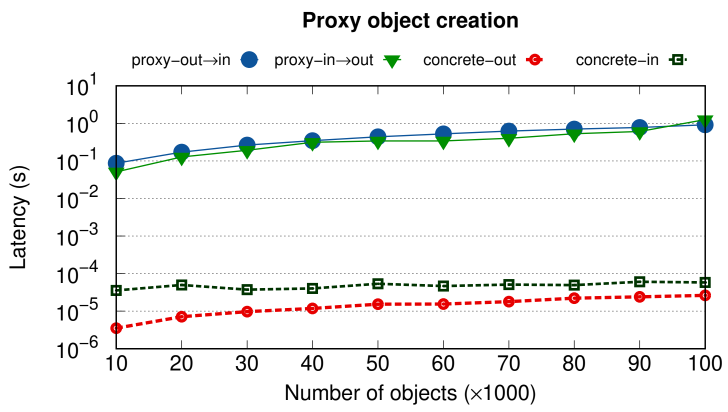


**Figure 3.3:** Performance of proxy object creation vs. concrete object creation.

## 3.5 EVALUATION

This section presents an experimental evaluation of MONTSALVAT based on micro- and macro-benchmarks with real-world applications. We seek to answer the following questions:

**Q1**: What is the cost of proxy object creation and remote method invocations? (§3.5.2, §3.5.3)
**Q2**: How does partitioning impact garbage collection performance? (§3.5.4)
**Q3**: How does partitioning impact application performance? (§3.5.5)
**Q4**: How do partitioned and unpartitioned native images in SGX enclaves compare with applications running on a JVM in enclaves? (§3.5.6)
**Q5**: What degree of TCB reduction is achieved by MONTSALVAT? (§5.6.4)

### *3.5.1 Experimental setup*

Our evaluation is conducted on a server equipped with a quad-core Intel Xeon E3-1270 CPU clocked at 3.80 GHz, and 64 GB of DRAM. The processor has 32 KB L1i and L1d caches, 256 KB L2 cache and 8 MB L3 cache. The server runs Ubuntu 18.04.1 LTS 64 bit and Linux kernel 4.15.0-142. We run the Intel SGX platform software, SDK and driver version v2.11. The EPC size on this server is 128 MB, of which 93.5 MB is usable by enclaves. The enclaves have maximum heap sizes of 4 GB and stack sizes of 8 MB. All native images are built with a maximum heap size of 2 GB.

We use SCONE to run unmodified applications on a JVM in SGX enclaves. The SCONE containers are based on Alpine Linux [2]. The base SCONE image ships OpenJDK8.[7] For non-SCONE experiments, we execute the GRAALVM compiler, which generates the native images, in OpenJDK-8u282. At execution, only the code of the generated native images runs. All reported latencies are averaged over 5 runs.

### *3.5.2 Performance of proxy creation*

*Q1: what is the cost of proxy object creation?*

We aim to evaluate the latency of proxy object creation in relation to normal (concrete) object creation. The concept of *proxy* is an intrinsic feature of MONTSALVAT, and it has the potential to influence application performance. We use a synthetic Java program to realise this experiment. Figure 3.3 shows the results obtained. We perform object instantiations in four different scenarios: concrete object

7 Tag: `8u181-jdk-alpine-scone5.1.0`

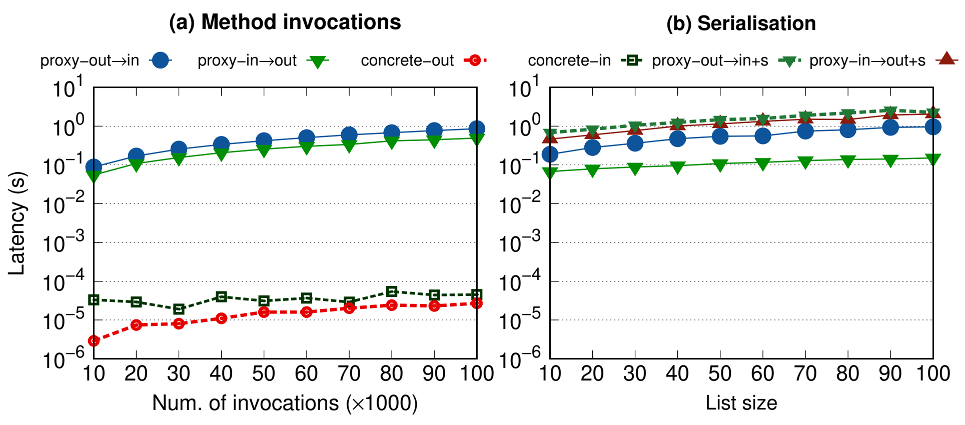


**Figure 3.4:** Performance of remote method invocations (RMIs) by proxy objects vs. concrete object invocations, and impact of serialisation on RMIs.

creation in and out of the enclave methods (respectively labelled `concrete-in` and `concrete-out` in Figure 3.3), and proxy object in (`proxy-in→out`) and out (`proxy-out→in`) of the enclave. Scenario `concrete-out` corresponds to the base line for this experiment.

We observe that proxy object creation latency in the enclave is 3 orders of magnitude higher when compared to concrete object creation in the enclave, and proxy object creation latency out of the enclave is 4 orders of magnitude higher when compared to concrete object creation out of the enclave. This performance drop when creating proxy objects is mainly due to the expensive enclave transitions required to instantiate the corresponding mirror objects in the opposite runtime.

### 3.5.3 *Performance of RMI and impact of serialisation*

*Q1: what is the cost of remote method invocations?*

The goal of this experiment is to study the performance of remote method invocations by proxy objects, and understand the impact of serialisation on these invocations. To this end, we generate synthetic programs where objects perform method invocations in four different scenarios: concrete object invoking instance methods in and out of the enclave (respectively labelled `concrete-in` and `concrete-out` in Figure 3.4 (a)) and proxy object invoking instance methods remotely from within (`proxy-in→out`) and out of (`proxy-out→in`) the enclave without serialisable parameters. We vary the number of method invocations of the objects in these scenarios and calculate the corresponding latency of the invocations.

Figure 3.4 (a) shows that when there is no serialisation involved, the latency of proxy object RMI in the enclave is 3 orders of magnitude higher than the latency of concrete object method invocation in the enclave. On the other hand, the latency of proxy object RMI out of the enclave is 4 orders of magnitude higher when compared to concrete object method invocation latency. This overhead is similar to that observed in proxy object creation, and is mainly due to the expensive enclave transitions involved.

To understand the impact of serialisation on proxy method invocations, we introduce two more scenarios where proxies in and out of the enclave invoke methods with a serialisable parameter (respectively labelled `proxy-in→out+s` and `proxy-out→in+s` in Figure 3.4). The serialised parameter is a list of 16 byte string values. We vary the size of the serialised list while keeping the number of method invocations constant at 10,000 invocations. For scenarios `proxy-in→out` and `proxy-out→in`, we use the same methods as in the `...+s` variants but without passing the list as

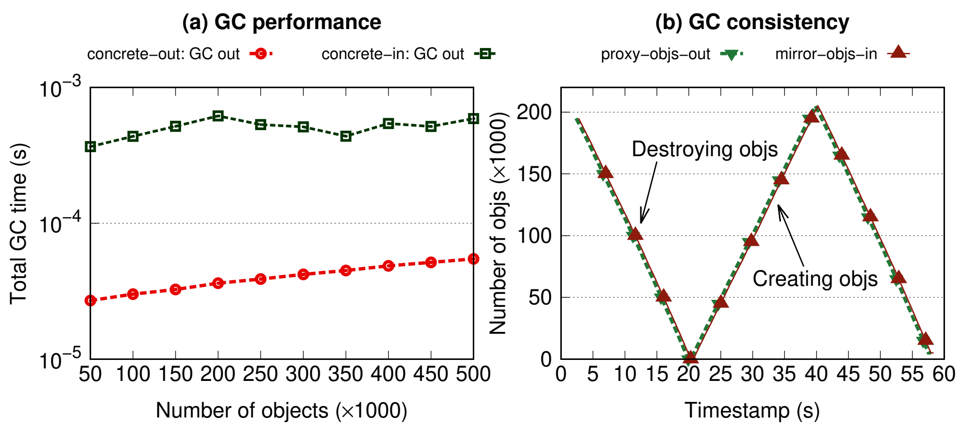


**Figure 3.5:** Garbage collection performance.

parameter.

Figure 3.4 (b) shows that RMIs in the enclave with the serialised parameter are about 10× more expensive than the corresponding RMIs without serialisation, while RMIs out of the enclave are about 3× more expensive than the corresponding RMIs without serialisation.

Although the overhead of RMIs observed is relatively high, it is important to recognise that the orders of magnitude and ratios calculated will vary based on the latency of the method operations themselves, without taking into account the cost of parameter serialisations or enclave transitions. The methods used in these experiments are setter methods updating an object field, which are relatively inexpensive operations. However, for more expensive methods, the cost of the method operations is likely to outweigh the cost of enclave transitions or parameter serialisations, hence decreasing the significance of enclave transitions on overall system performance.

### *3.5.4 Garbage collection performance*

*Q2: how does partitioning impact garbage collection performance?*

To understand the performance variations of garbage collection operations in and out of the enclave, we performed an experiment which involves creating multiple concrete objects, making them eligible for GC, and invoking the garbage collector in and out of the enclave. We record the total time spent for garbage collection in both scenarios. Figure 3.5 (a) shows the results obtained. We observe that the enclave adds an order of magnitude of overhead to the garbage collection operation. GraalVM native images embed a serial stop and copy GC [138]; the copy operation of this GC in the enclave leads to more data exchange between the CPU and the EPC, which explains the overhead when compared to GC performance outside (*i.e.*, `concrete-out`).

We performed a second experiment to demonstrate garbage collection consistency in and out of the enclave. In this experiment, a synthetic Java program creates proxy objects in the untrusted runtime, makes some of the objects eligible for GC and invokes the GC in the untrusted runtime. This operation is repeated for a given time range. The number of live (not garbage collected) proxy objects out of the enclave and the number of mirror objects in the enclave mirror-proxy registry are recorded at different timestamps. Figure 3.5 (b) shows the results obtained. We observe that as proxy objects are garbage collected, mirror objects are removed from the in-enclave mirror-proxy registry, making them eligible for GC too. In the same way, as more proxies are created, we notice a similar increase in the number of mirror objects in the enclave. These results suggest garbage collection consistency between the trusted and untrusted image.

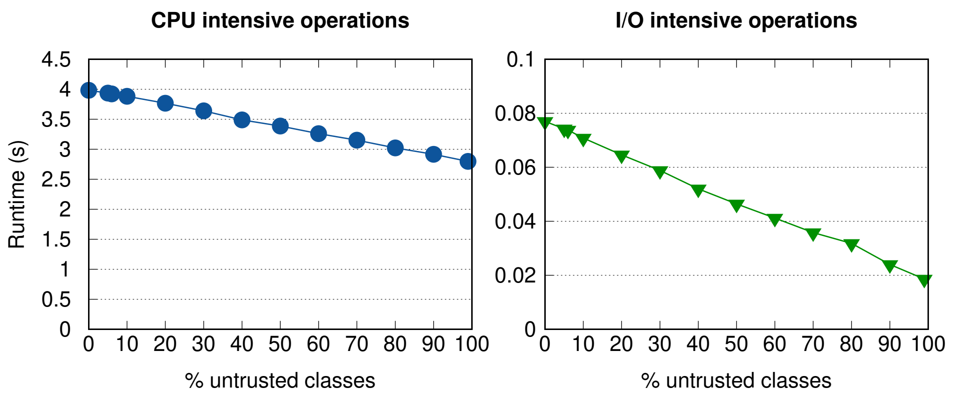


**Figure 3.6:** Enclave performance improves as classes are removed from the enclave.

### 3.5.5 *Impact of partitioning on application performance*

*Q3: how does partitioning impact application performance?*

To demonstrate the performance impact of partitioning native applications for enclaves with MONTSALVAT, we leverage a synthetic Java program, and two real-world applications, *i.e.*, PalDB [105] and GraphChi [93].

**Synthetic benchmark.** We developed a Java program generator to create Java applications with varying numbers of classes annotated as trusted or untrusted. We generated a Java application with 100 classes. Each class contains an instance method which performs either CPU intensive operations (*i.e.*, compute a fast Fourier transform [28] on a 1 MB double array) or I/O intensive operations (*i.e.*, writes 4 KB of data to a file). The main method instantiates each class and invokes the associated instance method. We vary the number of trusted and untrusted classes for two scenarios: *(1)* all class instance methods perform CPU intensive operations and *(2)* all class instance methods perform I/O intensive operations. We then calculate the total execution time of the resulting application. Figure 3.6 shows the results.

We observe that, as the percentage of untrusted classes increases (*i.e.*, more classes are moved out of the enclave), the total run time of the application decreases. For I/O operations, fewer enclave transitions are done for I/O write operations, leading to better performance. For CPU operations, enclave performance can get more expensive when random reads and writes are done on data which is not present in the CPU [7, 210]. This decrease in performance is caused by on-the-fly encryption/decryption of CPU cache-lines by the MEE [210] when data is transferred between the CPU and the EPC. This synthetic benchmark suggests that by delegating (non-sensitive) computations to the untrusted runtime, we relieve the enclave of expensive cryptographic operations, leading to better enclave performance, and better overall application performance. We illustrate this further with the two real world applications below.

**PalDB.** PalDB is an embedabble persistent key-value store developed by LinkedIn, used in analytics workflows and machine-learning applications. We consider a Java application based on PalDB which writes and reads a list of key-value (K/V) pairs in a store file. The keys are string values of randomly generated integers (in the range $[0, 2^{31} - 1]$), while the values are randomly generated strings of length 128. For this, we introduced two classes: `DBReader` and `DBWriter`[8] which exploit PalDB's API for respectively reading from and writing to the store file.

8 PalDB DBReader and DBWriter

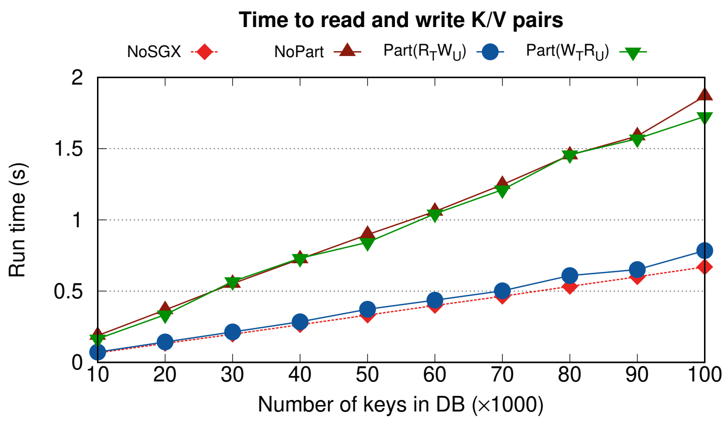


**Figure 3.7:** Read and write times for partitioned PalDB.

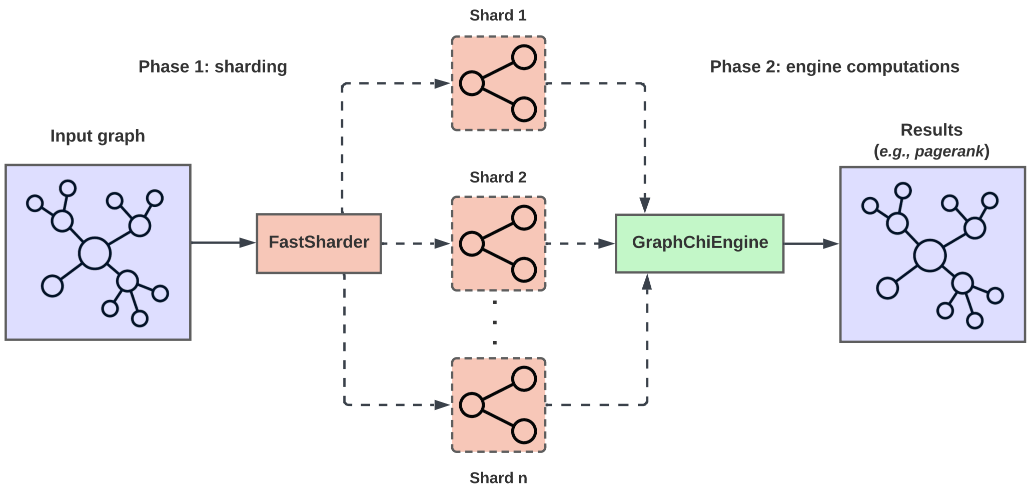


**Figure 3.8:** Typical GraphChi program workflow.

A natural and intuitive partitioning approach for this application involves partitioning along the `DBWriter` and `DBReader` classes, depending on the security requirements of the application. For this we consider two possible scenarios: `DBReader` trusted and `DBWriter` untrusted ($R_T W_U$), and `DBReader` untrusted and `DBWriter` trusted ($R_U W_T$). We run the unpartitioned application (base line) as a native image in an SGX enclave and compare its performance to the partitioned version with the above mentioned schemes, as well as the native image running without SGX enabled. Figure 3.7 shows the results obtained.

For both partitioning schemes ($R_T W_U$ and $R_U W_T$) we observe performance improvements after partitioning the application. $R_T W_U$ is on average 2.5× times faster while $R_U W_T$ is on average 1.04× faster when compared to the unpartitioned native image. PalDB optimises reads by memory mapping the store file in memory, but does regular I/O for writes to the store file. This explains the greater performance improvement after partitioning using $R_T W_U$ as it relieves the enclave of expensive write-induced enclave transitions to perform I/O, making it closer to the native performance (no SGX). For $R_U W_T$ the performance improvement is less as more `ocall` transitions (23× more on average compared to $R_T W_U$) are done to write K/V pairs to the store file from within the enclave. As expected, the application has best performance when running without SGX, however this is the most insecure configuration.

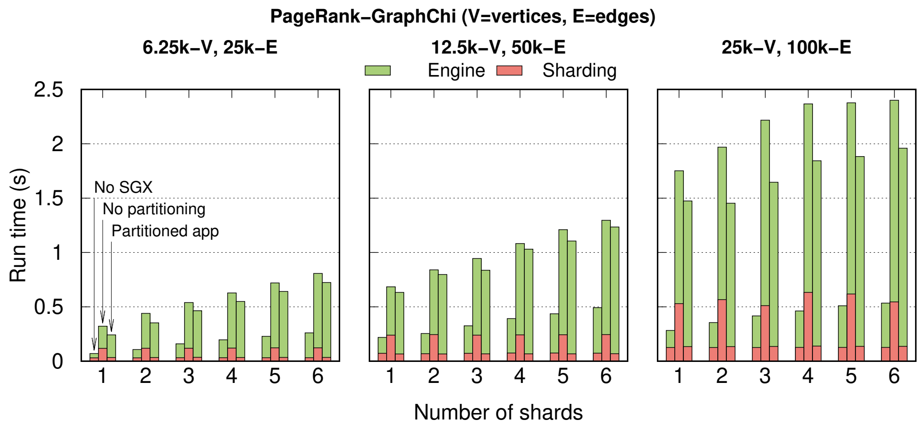


**Figure 3.9:** Execution time for partitioned vs. unpartitioned GraphChi.

**GraphChi.** GraphChi is a large-scale graph processing framework. We use the popular *PageRank* [3] algorithm as an example application for partitioning. PageRank evaluates the relative importance of nodes in a directed graph. Typically, GraphChi applications follow the workflow outlined in Figure 3.8. The input graph is split by a sharder (`FastSharder`) into multiple parts (shards) which are then processed in the core execution engine (`GraphChiEngine`) to produce the final result (PageRank values in our case). A possible partitioning scheme for the application would be along the `FastSharder`[9] and `GraphChiEngine`[10] classes. For this we make the `GraphChiEngine` trusted and the `FastSharder` untrusted. We run the PageRank algorithm on synthetic directed graphs generated using the RMAT algorithm [17]. We vary graph sizes by varying the number of vertices ($V$) and edges ($E$) in the graph. For each graph, we vary the number of shards for the PageRank computations and compare the performance of the partitioned native image to the unpartitioned case, as well as the native image running without SGX. Figure 3.9 shows the results obtained.

For each shard, the leftmost bar shows the run time without SGX, the middle bar for the unpartitioned native image running inside the enclave, and the rightmost bar for the partitioned application. We show the processing time for the PageRank itself (executed in GraphChiEngine) of the graph nodes, as well as the portion of the total time spent on graph sharding (in FastSharder).

After partitioning, the `FastSharder` is transferred to the untrusted runtime, relieving the enclave of all expensive I/O related work done by the sharder, thereby improving enclave latency. We can observe on the graph that the latency due to sharding after partitioning is approximately the same as the native case (no SGX), which is explained by the fact that the `FastSharder` now operates in the untrusted runtime, thus no overhead due to MEE encryption/decryption operations in the enclave. This leads to a performance gain of about 1.2× on average as compared to the unpartitioned case. We observe similar performance improvements for the different graph sizes.

**Take-away 1**

Partitioning improves in-enclave application performance. This is attributed to a decrease in enclave transitions for I/O operations and the elimination of overhead from SGX-related cryptographic operations for the removed code.

9 FastSharder class
10 GraphChiEngine class

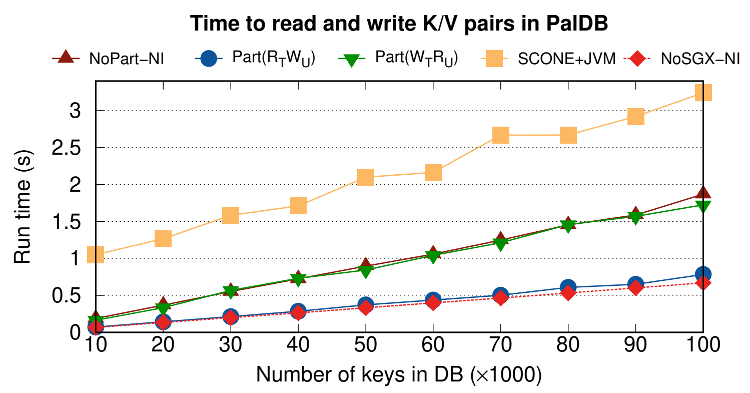


**Figure 3.10:** Partitioned and unpartitioned PalDB native images vs. PalDB in SCONE+JVM.

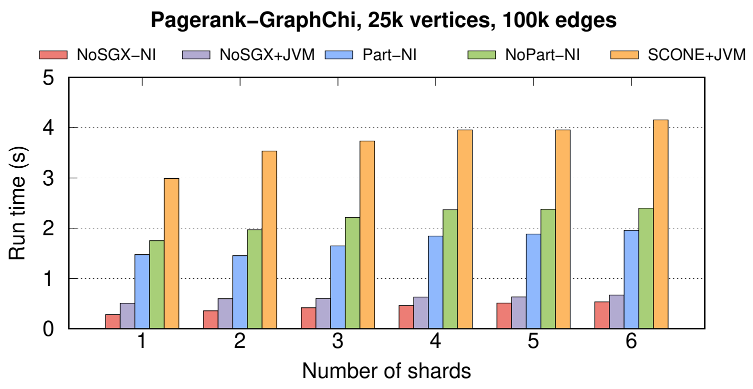


**Figure 3.11:** Partitioned and unpartitioned GraphChi native images vs. GraphChi in SCONE+JVM.

### 3.5.6 *Comparing JVM-based applications in enclaves*

*Q4: how do partitioned and unpartitioned native images in SGX enclaves compare with applications running on a JVM in enclaves?*

To understand how partitioned and unpartitioned native images compare to their JVM-based counterparts running in enclaves, we compared SGX-based native images to the applications running on a JVM in a SCONE container. The JVM is run with a maximum heap size of 2 GB (`-Xmx2G`). We are not particularly concerned with the performance of applications running on a JVM out of enclaves. However, we included results for the latter to get a clearer picture of the performance variations using the different approaches.

**Partitioned native images vs. JVM-based applications in enclaves.** For this experiment we compared the partitioned versions of PalDB and GraphChi, using the same partitioning schemes, to their unpartitioned counterparts running on a JVM in a SCONE container. Figures 3.10 and 3.11 show the results for PalDB and GraphChi respectively.

From the results, we observe that $R_T W_U$ and $R_U W_T$ are respectively 6.6× and 2.8× faster on average when compared to PalDB running on a JVM in SCONE. As for GraphChi, the partitioned GraphChi native image is 2.2× faster on average when compared to GraphChi running on a JVM in SCONE.

Two main factors account for the poorer performance of applications running on a JVM in SCONE: *(1)* the JVM spends some time for class loading, bytecode interpretation, and dynamic compilation; these operations are absent in native images, *(2)* the in-enclave JVM increases the number objects in the enclave heap, which leads to more data exchange between the EPC and CPU, hence more

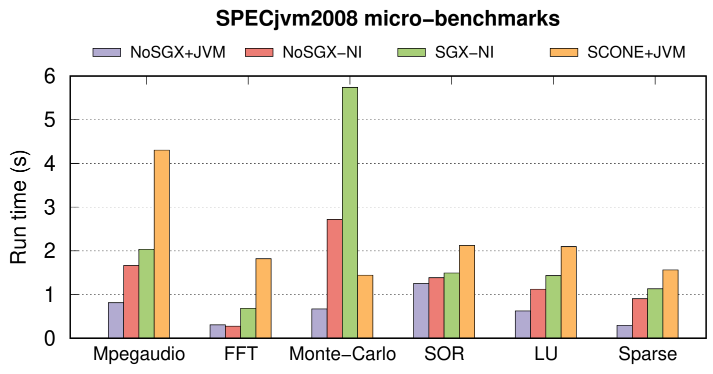


**Figure 3.12:** Performance of unpartitioned SPECjvm 2008 micro-benchmarks in enclaves.

expensive MEE encryption/decryption of CPU cache lines when compared to the native images in the enclaves where class loading, bytecode interpretation, and dynamic compilation are absent at runtime.

**Unpartitioned native images in enclaves vs. JVM-based applications in enclaves**. Here we compare the performance of unpartitioned native images in enclaves to their JVM-based counterparts. We present results for PalDB (Figure 3.10) and GraphChi (Figure 3.11), as well as 6 SPECjvm2008 [179] micro-benchmarks (Figure 3.12) using their default workloads.

The unpartitioned PalDB and GraphChi native images in the enclave are respectively 2.6× and 1.7× faster when compared to their JVM counterparts in a SCONE container. We observe comparatively lower performance for the JVM counterparts in a SCONE container except for the Monte-Carlo micro-benchmark. This could be attributed to garbage collection cycles triggered in the native image. Recent studies [84] suggest the GC in GraalVM native-image performs poorly when compared to OpenJDK HotSpot JVM's garbage collectors. The poorer in-enclave JVM results for the rest can be justified by the two aforementioned reasons.

### *3.5.7 TCB analysis*

*Q5: what degree of TCB reduction is achieved by* MONTSALVAT*?*

We conduct a TCB analysis of our system along two dimensions. First, we compare the TCB of MONTSALVAT to a LibOS-based approach which includes the JVM inside the enclave. We evaluate the number of lines of code for the in-enclave components for both approaches. We recall that MONTSALVAT leverages GraalVM's SubstrateVM (SVM) [134] to provide runtime components for the AOT compiled native images. The LibOS-based approach relies on Gramine (formerly Graphene-SGX) v1.1 [163] and OpenJDK8 JVM. The results are outlined in Table 3.1. We observe that MONTSALVAT achieves over 9× reduction in TCB (in LoC) with respect to a LibOS-based system where the LibOS, JVM, and libc are included inside the enclave.

Subsequently, for each partitioned application, we evaluate the size of the in-enclave components before and after partitioning with Montsalvat. This further reduces the TCB by excluding untrusted classes from the enclave. For PalDB, the in-enclave application code is reduced from 3140 to 3070 LoC, while for GraphChi, the in-enclave application code is reduced from 9760 to 8507 LoC. The extent of TCB reduction here is entirely dependent on the application being partitioned and varies based on the size and number of classes marked as untrusted.

**Table 3.1:** Size in LoC of in-enclave components for MONTSALVAT and a LibOS-based system.

| Component | ≈ LoC | MONTSALVAT | LibOS |
|---|---|---|---|
| SubstrateVM | 210,780 | ✓ | ✗ |
| Shim library | 11,600 | ✓ | ✗ |
| JVM | 593,159 | ✗ | ✓ |
| JNI runtime libraries | 423,303 | ✗ | ✓ |
| Gramine | 40540 | ✗ | ✓ |
| GNU libc 2.19 | 1,008,780 | ✗ | ✓ |
| **Total LoC** | | 222,380 | 2,065,782 |

**Take-away 2**

MONTSALVAT achieves over 9× TCB reduction compared to a LibOS-based approach which incorporates the entire JVM and libc within the enclave.

### 3.5.8 *Additional use-case scenarios*

MONTSALVAT can be used for a wide variety of security applications. Examples include secure key/value stores and blockchain applications. The classes/business logic for storing and retrieving key/value pairs, and business logic for smart contracts can be secured in the enclave, while classes for network-related functionality are kept out of the enclave. The partitioned components then interact in accordance with our design.

## 3.6 RELATED WORK

We classify the related work into four categories: *(i)* systems that make it possible to run full, unmodified applications inside enclaves, *(ii)* framework-specific systems that support partial execution inside enclaves, *(iii)* systems that allow for partitioning generic native applications, and *(iv)* systems that allow for partitioning generic Java applications.

**Running full applications inside enclaves.** Systems such as Haven [11], SCONE [7], Graphene-SGX [195] and SGX-LKL [160] propose solutions to run entire legacy applications inside enclaves. They introduce a library OS into the enclave to emulate OS logic. For instance, SCONE leverages a modified version of the *libc* to run microservices inside Docker [38] containers. While these solutions offer good compatibility with a wide range of applications and require low developer effort, they introduce hundreds of thousands of lines of code into the TCB. This leaves more room for security vulnerabilities and may significantly decrease enclave performance.

Similar to the above tools, EnclaveDom [114] maintains the entire application inside the enclave, but partitions the enclave into tagged memory regions so as to achieve privilege separation. This strategy does not achieve TCB reduction per se, but mitigates the security issues linked with large-TCB applications.

**Framework-specific partitioning.** Some recent systems propose to manually partition specific frameworks and/or the applications that run on them into trusted and untrusted parts. VC3 [170] is a system for trustworthy data analytics in Hadoop that requires manually rewriting Map and Reduce functions to be used in SGX enclaves, while keeping the main Hadoop library outside the enclave. SecureKeeper [14] proposes an extension to ZooKeeper which preserves confidential user data inside enclaves while maintaining the ZooKeeper framework outside the enclave. Plinius [220] manually partitions a persistent memory and machine learning (ML) library in order to enable efficient ML in SGX enclaves. Opaque [228] is a secure data analytics platform built on top of Spark SQL that focuses on preventing access pattern leakage; it notably introduces SGX-enabled oblivious operators that can be used on tables that store sensitive data. While these systems reduce the size of the TCB, they focus on individual frameworks, limiting their flexibility.

**Native code partitioning.** Glamdring [104] provides a technique to automatically partition C/C++ applications into trusted and untrusted parts using static program slicing. Panoply [177] introduces *micro-containers* which expose standard POSIX abstractions and run inside enclaves; applications must be refactored to extract sensitive code and data to be placed inside micro-containers. These systems do not tackle the complexities introduced by managed languages.

**Java code partitioning.** Civet [197] and Uranus [79] are two recent frameworks for running parts of Java applications inside enclaves. Civet requires defining methods which will serve as the partitioning boundary, and Uranus requires annotating all sensitive methods. We argue that placing the enclave boundary at class level is more intuitive for developers. Moreover, our system benefits from GraalVM's optimisations, such as class initialisations at build time, rather than at runtime. Furthermore, our approach relies on a small shim library to relay unsupported calls to the untrusted runtime. Civet on the other hand maintains a full library OS (Graphene SGX [195]) inside the enclave, leading to a larger TCB as previously shown. CFHider [206] also proposes to run parts of Java applications inside enclaves, but it specifically focuses on branch statement conditions with the objective to guarantee control flow confidentiality.

## 3.7 SUMMARY

This chapter introduced Montsalvat, a tool to partition Java applications for enclaves. Montsalvat leverages code annotations to specify sensitive application classes, and bytecode transformations to partition the application into trusted and untrusted components, based on the code annotations. Montsalvat enables object communication between the partitioned components, and ensures consistent garbage collection across the trusted and untrusted application heaps. We show the practicality of Montsalvat with real-world applications, and our extensive evaluations show TCB reduction can be achieved while improving the application's performance.

**Prospects.** While Montsalvat performs partitioning along class boundaries, there is potential for extending this to the method level. This entails specifying methods as either trusted or not. Similarly, the application classes will have two versions, one residing within the enclave and the other out of the enclave. Each class will contain a combination of fully-implemented methods or proxy methods which perform transitions to the opposite runtime. In the next chapter, we will address multi-language program partitioning at the function/method level.

CHAPTER 4

# Multi-language Program Partitioning for Enclaves

The previous chapter discussed how to tackle the partitioning challenge specifically for Java, a widely-used high-level programming language. This chapter extends the ideas proposed to encompass a multi-language context, with the goal of building a more generic partitioning tool, applicable to a broader range of programming languages. In that light, it proposes SECV, a multi-language technique to analyse and partition programs for TEEs like Intel SGX. SECV leverages GraalVM's TRUFFLE framework, which provides a language-agnostic abstract syntax tree (AST) representation for programs, to provide special AST nodes called *secure nodes* that encapsulate sensitive program information. It then employs dynamic taint-analysis to analyse and partition applications based on the secure nodes.

This chapter is organised as follows:

## 4.1 INTRODUCTION

Various partitioning tools have been proposed in order to minimise the TCB of TEE-based applications. However, the proposed tools all target specific languages: Montsalvat (Chapter 3), Civet[197], and Uranus[79] for Java, GOTEE [49] for Go, and Glamdring [104] for C. These tools depend heavily on the language semantics and cannot be readily reused for other languages. To support the next programming language (*e.g.*, Python, R, JavaScript, Ruby, *etc.*), one must entirely re-implement a new automatic partitioning tool, which is time consuming, error-prone, and expensive.

In this work, we propose *to decouple the partitioning tool from the language semantics, and to implement the former once and for all languages.* To that end, we leverage TRUFFLE [212], a Java library for building high-performance language interpreters. It provides a generic abstract syntax tree (AST) composed of nodes that represent various syntactic elements of a program: *i.e.*, expressions (*e.g.*, function calls or arithmetic operations), program values (*e.g.*, literals or variables), control flow (*e.g.*, if-else or for loops), *etc.* TRUFFLE provides support for popular programming languages like Python, JavaScript, R, Ruby, C/C++, amongst many others. To implement the partitioning tool once and for all, we first introduce a new multi-language AST node type, called a *secure node*. This node contains a *secure value* corresponding to a sensitive value that has to be secured in the enclave. Then, we leverage the *polyglot interoperability protocol* provided by TRUFFLE [57], which allows the declaration of a secure value from any language with an expression as simple as `x = polyglot.eval("secV", "sInt(42)")`. Finally, we develop a generic *dynamic taint tracking* tool, POLYTAINT, which instruments TRUFFLE ASTs to track the data flow of sensitive values from secure nodes so as to determine the portions of a program (*e.g.*, *functions*) to be shielded inside the enclave. POLYTAINT then partitions the program into a trusted and untrusted component, to be executed respectively inside or outside the enclave.

In summary, we propose the following contributions:

1. Generic AST secure nodes to specify sensitive data in any TRUFFLE language.
2. POLYTAINT, a TRUFFLE instrumentation tool which performs dynamic taint tracking on generic polyglot programs and partitions the programs based on the use of secure values.
3. An extensive experimental evaluation demonstrating the effectiveness of our approach via micro-benchmarks in JavaScript and Python, as well as real-world applications: *PageRank* [3] and *linear regression* [216]. Our analysis of partitioned programs shows we can reduce the size of the TCB (*i.e.*, improved security) and improve performance (up to 14.5%) at the same time.

## 4.2 BACKGROUND

**TRUFFLE nodes.** As previously described in Chapter 2 (§2.2.1), TRUFFLE is a framework provided by the GraalVM ecosystem to build tools and programming language implementations as self-modifying AST interpreters [153, 57]. At low level, TRUFFLE provides a base *Node* class that is leveraged by language implementers to build other AST nodes representing the semantic constructs of their programming language, *e.g.*, an addition operation, a variable write, *etc.* Essentially, every node in a TRUFFLE AST is executed (via an `execute` method) at runtime to produce a result.

In Figure 4.1 for example, the integer addition node is executed at runtime to produce the sum of the left and child node values, which are computed by calling the *execute* methods of the left and right nodes respectively.

The key advantage of TRUFFLE is that all programs implemented in a supported language are parsed to a common AST representation (*i.e.*, the TRUFFLE AST), which is then manipulated in a language-agnostic fashion. TRUFFLE languages include JavaScript (JS) [136], Ruby [156], R [132], Python

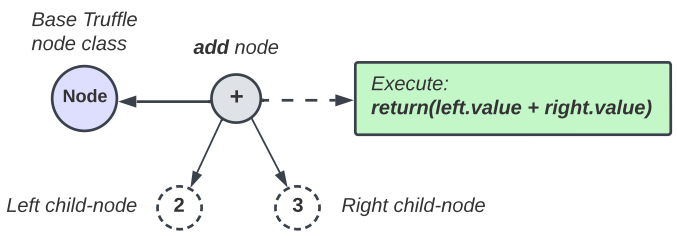


**Figure 4.1:** Integer addition node executed in the AST to produce the sum of the values of left and right child.

[135], LLVM-based languages [150], and more.

**TRUFFLE polyglot API.** TRUFFLE allows developers to build *polyglot* applications that combine code written in different languages. This interoperability is provided by the *polyglot interoperability protocol*, a set of standardised messages (*polyglot API*) implemented in every TRUFFLE language [57, 147]. This API enables the transfer of objects between different language scopes as TRUFFLE *values*, *i.e.*, an instance of `Value` class.

```java
import org.graalvm.polyglot.*;
class PolyglotTest {
  public static void main(String[] args) {
    Context polyglot = Context.create();
    Value array = polyglot.eval("js", "[1, 2, 3, 44]");
    int result = array.getArrayElement(3).asInt();
    System.out.println(result);  // prints 44
  }}

```

**Listing 7:** Java application accessing an object from a JS language scope via the polyglot API.

Listing 7 shows the transfer of an array object created in a JS language scope into the Java language scope via the polyglot API (line 5). The first parameter to the `polyglot.eval` call is the TRUFFLE *language id*, a unique string identifier for TRUFFLE guest languages, while the second parameter is the guest source code to execute. Such cross-language object exchange is possible in all TRUFFLE languages, which simplifies the implementation of multi-language applications.

Henceforward in this text, the language accessing the objects generated by another interpreter will be referred to as the *host language* (Java in Listing 7), and the language used to generate the objects accessed by the host will be the *guest language* (JS in Listing 7). Objects of the host language will be referred to as *regular objects* while those of the guest language will be referred to as *foreign objects*.

### *4.2.1* TRUFFLE *instrumentation agents*

In the context of ASTs, *instrumentation* refers to the process of adding additional behaviour or information gathering features to a program. As a result, program instrumentation can be leveraged to effectively monitor and trace the flow of sensitive information across a program, and take appropriate action. TRUFFLE framework offers vast support for program instrumentation, and provides an API to dynamically intercept the execution of nodes in an AST [199, 57, 213]. This API has been used to implement tools like a program profiler [139], a debugger [137], and a taint-tracking tool [92]. In POLYTAINT, we leverage this API to implement our partitioning tool.

To intercept the execution of nodes, the developer has to implement an *instrumentation agent* [199]. The agent intercepts the execution of nodes by leveraging *syntactic tags* associated to the AST nodes. These tags give the semantics of the nodes (*e.g.*, *call* tag, *variable write* tag *etc.*).

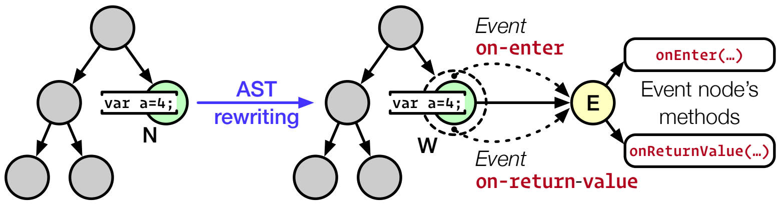


**Figure 4.2:** Instrumenting a variable write with a wrapper node.

Figure 4.2 illustrates how an instrumentation agent works. When the agent is loaded, it associates an *event node* to a tag [131]. In our example, the agent attaches the event node $E$ to any AST node with the *variable write* tag. At execution, when TRUFFLE visits an AST node with the instrumented tag for the first time, it replaces the node with a wrapper node [154] connected to the event node. In our example, when TRUFFLE visits the node $N$ for the first time, it wraps $N$ in $W$ and connects $W$ with $E$.

A wrapper node implements a special `execute` function. Upon invocation, it first calls `onEnter` on the associated event node, then `execute` on the wrapped node, and finally `onReturnValue` on the associated event node. These functions can collect metrics, modify the wrapped node, and even replace the wrapped node by another node. For example, the event node's `onReturnValue` method could be used to register the name of the variable being written, as well as the corresponding value.

*Key Insight*

SECV provides special *secure nodes* to encapsulate sensitive information. **Any TRUFFLE language** can contain these secure nodes via the polyglot API. The resulting AST is instrumented to determine parts of a program, *e.g.*, variables, methods, classes, that access sensitive data, and hence should be isolated in a secure enclave.

## 4.3 THREAT MODEL

We consider a powerful adversary with full control over the software stack, including privileged software (*i.e.*, host OS, hypervisor) with access to the physical hardware, *i.e.*, DRAM, secondary storage, *etc*. The adversary's goal is to disclose sensitive data or damage its integrity.

SECV's workflow assumes enclave program development, taint tracking, and program partitioning, as well as final enclave code building and signing are done in a trusted environment. This prevents malicious code tampering aimed at disclosing sensitive information at runtime. The integrity of the trusted partition can be ensured via remote attestation.

We assume the adversary cannot open the CPU package to extract decrypted enclave secrets, and that the final enclave code does not intentionally leak sensitive information (*e.g.*, encryption keys). We do not consider denial-of-service (DoS) and side-channel attacks [16, 171], for which mitigations exist [60, 128].

## 4.4 DESIGN AND WORKFLOW OF SECV

SECV is a framework that analyses applications and partitions them into *trusted* and *untrusted* parts for TEEs. It supports applications written in *any* Truffle language, and introduces *secure values* to specify sensitive data. Figure 6.5 presents an overview of SECV's workflow, which comprises 4 phases:

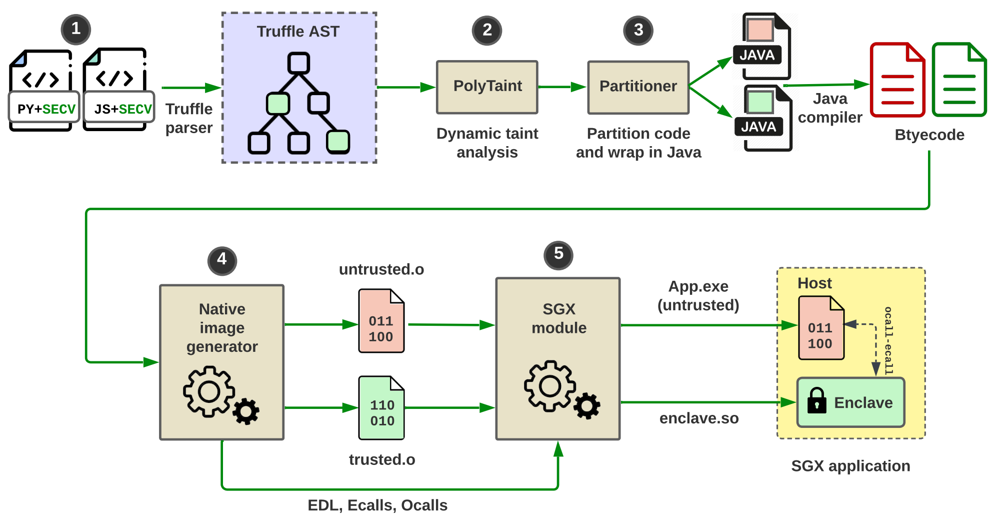


**Figure 4.3:** Overview of SECV's workflow.

*(❶)* identification of sensitive variables that contain *secure values* (see §4.4.1), *(❷)* dynamic taint tracking with POLYTAINT (see §4.4.2 and §4.4.3), *(❸)* program partitioning with the *partitioner* (see §5.4.5), and *(❹–❺)* native image building with the *image generator* and final application creation (see §4.4.5).

We illustrate SECV's workflow by considering a simple linear regression program written in Python (see Listing 8). We consider a security scenario where we wish to keep the learned model (*i.e.*, `m` and `c`) confidential [72, 164, 223, 122]. In the following, we show how one can use SECV to enforce this scenario.

```
import polyglot
m = 0.0
c = 0.0
L = 0.0001
N = 10000
numIter = 100

def readXData(n):
  # logic (omitted)
  return xData

def readYData(n):
  # logic (omitted)
  return yData

def arraySum(A):
  sumA = 0
  for a in A:
    sumA += a
  return sumA
def trainModel(numIterations):
  X = readXData(N)
  Y = readYData(N)
  for i in range(numIterations):
    Y_pred = m * X + c
    D_m = (-2/N) * arraySum(X * (Y-Y_pred))
    D_c = (-2/N) * arraySum(Y-Y_pred)
    m = m - L * D_m
    c = c - L * D_c

trainModel(numIter)
```

**Listing 8:** Illustrative example: simple linear regression program written in Python.

## 4.4.1 *Identifying and specifying sensitive data*

We propose a multi-language approach to specify sensitive data via the use of newly-introduced *secure nodes* in a TRUFFLE AST. All values associated with these secure nodes are referred to as *secure values* and they represent sensitive information which must be shielded within the enclave. To make this possible, we leverage the TRUFFLE framework to build a *secure node generator* which injects secure nodes into any TRUFFLE AST via the polyglot API. This secure node generator is implemented like a regular TRUFFLE language but provides special AST nodes which comprise secure information.

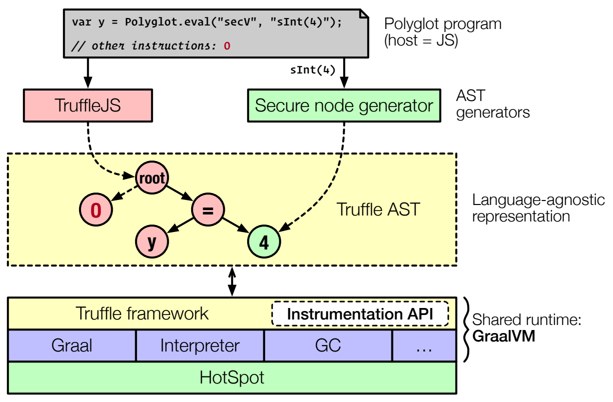


**Figure 4.4:** The TRUFFLE framework can be leveraged to provide an AST generator that produces secure nodes which can be injected in any TRUFFLE AST via the polyglot API.

Our preliminary prototype provides secure node types: `sInt`, `sDouble`, `sBoolean`, and `sArray` to contain secure int values, secure double values, secure boolean values, and secure array object values respectively. For example, Listing 9 shows how a JavaScript program can inject a secure integer value into its AST.

```
var myInt = 2;
var secInt = Polyglot.eval("secV", "sInt(4)");
myInt = secInt + 2;
console.log(myInt); // prints 6
```

**Listing 9:** Injecting a secure integer node with value 4 into a JS program via the polyglot API.

Going back to our illustrative example, since our goal is to secure our ML model (represented by `m` and `c`), we can specify `m` and `c` as secure values using the TRUFFLE polyglot API for Python. To do this, we change the variable assignments for `m` and `c` to the code shown in Listing 10 (lines 2 and 3).

```
2 m = polyglot.eval(language="secV", string="sDouble(0.0)")
3 c = polyglot.eval(language="secV", string="sDouble(0.0)")
```

**Listing 10:** Specifying m and c as secure values using SECV.

At runtime, the TRUFFLE AST corresponding to the Python program will have *SecV* nodes associated with `m` and `c` once the `polyglot.eval` call is executed. SECV then analyses the program's AST to determine which portions of the program access the secure values `m` and `c`.

**Support for other secure node types.** Implementing more secure types, *e.g.*, lists, maps, *etc.* in SECV is straightforward. It involves: defining a new TRUFFLE node for the type, implementing its internal representation, defining type operations and conversions, and integrating the new type node into SECV's language grammar.[1] To simplify the development of complex types, TRUFFLE's design allows the use of existing Java types, *e.g.*, arrays, sets, *etc.* as building blocks when defining the new node.

### *4.4.2 Taint tracking*

SECV comprises POLYTAINT, a TRUFFLE code instrumentation tool that supports applications written in any TRUFFLE language. POLYTAINT is designed as a taint tracking tool. It monitors the creation of secure nodes in a polyglot application and marks a node that uses a value from a secure node (*i.e.*, secure value) as secure itself.

**Assumptions.** Our preliminary implementation of SECV assumes a *procedural program structure* where the application to be partitioned is organised as a group of $n$ functions: $f_1, f_2, \ldots, f_n$. Most

1 SecV language grammar

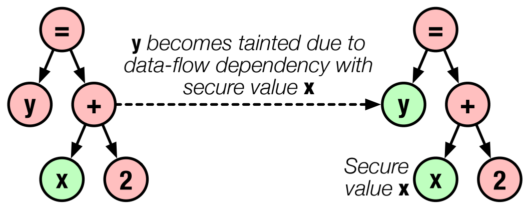

y becomes tainted due to data-flow dependency with secure value x
=
y
+
x
2
Secure value x

**Figure 4.5:** Taint propagation due to explicit data-flow dependencies with secure values.

programming languages support this paradigm. The goal of SecV is thus to determine the set of program functions to be put inside the enclave, and those to be kept out of the enclave, and partition the program into two parts based on this information. We assume that the inputs used in dynamic analysis are sufficiently exhaustive to cover all legitimate production configurations; this mitigates leakage of sensitive data at runtime due to incomplete code coverage during analysis.

#### 4.4.2.1 *Taint propagation*

In the context of our work, *taint propagation* defines how secure values flow from the secure nodes into subsequent parts of the program. PolyTaint performs taint propagation via *explicit information flow* [24]. That is, an untainted variable `y` becomes tainted if an already tainted variable `x` is directly involved in the computation of `y`'s value. In other words, there is a direct data-flow dependency between `x` and `y`. For example, the statement `y = x + 2` marks `y` as tainted if `x` is tainted, as shown in Figure 4.5. PolyTaint extends this idea to functions by marking them as tainted if tainted variables are manipulated in their bodies.

PolyTaint implements taint propagation via *AST instrumentation*. This is done by wrapping variable reads/writes, function calls (see §4.4.3), *etc.* and testing for nodes that access secure values. For example, in our illustrative example (Listing 10), the variable assignments at lines 2 and 3 are wrapped and evaluated by PolyTaint to obtain the corresponding variables (*i.e.*, `m` and `c`) that receive secure values (foreign objects in Truffle terminology) from secure nodes. We refer to such variables as *taint sources*.

**Program function classification.** The primary goal of taint propagation in PolyTaint is to determine which program functions access secure values (directly or indirectly from the taint sources) at runtime, and which functions do not. As such, PolyTaint classifies functions into three categories: *trusted*, *untrusted,* and *neutral*.

**Trusted functions.** These are functions that manipulate secure variables explicitly within their bodies, *i.e.*, instantiate secure variables (taint sources) or modify the values of secure variables. We define a *secure variable* as a program variable which explicitly receives its value from a SecV node via the polyglot API (*e.g.*, `m` and `c` in Listing 10) or a program variable which gets tainted following PolyTaint's taint propagation rules. As such, function `trainModel` in Listing 8 will be tagged as a trusted function by virtue of the variable write: `Y_pred = m * X + c` which uses secure variables `m` and `c` inside the function `trainModel`.

Trusted functions are included fully inside the trusted/enclave part of the partitioned application. These functions have *proxy functions* in the untrusted part. We define the *proxy function* of a program function `f` as a function which has the same signature as `f` but whose body is stripped and replaced with an enclave transition (*i.e.*, `ecall` or `ocall`) transfering execution control to `f`. Following this logic, the proxies of *trusted* functions perform `ecall` transitions. The primary rationale for this design choice is *security*: that is, secure variables that contain sensitive information cannot be accessed in the untrusted side.

**Neutral functions.** These are functions which access secure variables *only* through their input parameters and do not explicitly access already tainted (*i.e.*, not considering the tainted inputs) secure variables within their bodies. In other words, a neutral function does not access a tainted value when none of its arguments is tainted, and may access tainted values if at least one of its arguments is tainted. In Listing 8 for example, `arraySum` is considered a neutral function by virtue of the tainted inputs `Y - Y_pred` and `X * (Y - Y_pred)`. Moreover, untrusted code could also perform array sums without sensitive inputs.

Neutral functions are included fully in both the trusted and untrusted partitions; there are no proxy functions involved. The rationale behind this design choice is twofold: the approach allows for good *security* and *performance*. That is, enclave code can access the functionality of the neutral function without leaking sensitive information, via input parameters, to the outside (security), and the code out of the enclave does not need to perform an expensive `ecall` transition to access the functionality of the neutral function (performance). Utility functions such as those that calculate generic sums, products, *etc*. are examples of neutral functions.

**Untrusted functions.** These are functions which do not have any access to secure variables during the lifetime of the program. Untrusted functions are included fully only in the untrusted partition. However, they have *proxies* in the trusted side which perform `ocall` transitions to access the functionality in the untrusted side of the partitioned application. The rationale for this design choice is *TCB reduction*, and thus *security*, *i.e.*, code which is not sensitive need not be in the enclave following the principle of least privilege [167].

### *4.4.3 AST instrumentation*

To identify the trusted ($T$), neutral ($N$), and untrusted ($U$) functions, PolyTaint uses a dynamic analysis technique. It executes the program once during the development phase. During this execution, it identifies the secure values, and then deduces the state of the functions, *i.e.*, *trusted*, *untrusted*, and *neutral*. During the in-vitro execution, PolyTaint performs taint propagation by leveraging the Truffle instrumentation framework, which makes it possible to intercept the execution of AST nodes. To instrument the Truffle AST, PolyTaint intercepts the executions of these node types: *variable read/write*, *object field read/write*, *array element read/write* and *function call* nodes. PolyTaint provides an event node class for each of the node types to be wrapped,[2] as well as a base event node class `PolyTaintNode` which all the event node classes inherit from. `PolyTaintNode` implements common functionality which is used by the event node classes.

PolyTaint maintains a map data structure called the *taint map* which tracks program symbols (*e.g.*, variables, functions, *etc*.) that access secure values. A unique *string identifier* is associated to each program symbol such as *e.g.*, a variable or function. The *taint map* associates *string identifiers* to *taint labels*. We have 3 *taint labels*: 1 for tainted/trusted nodes, 2 for neutral nodes, and 0 for untrusted nodes (*i.e.*, representing untrusted functions).

During instrumentation, PolyTaint leverages the `onReturnValue` callback of event nodes to test for tainted program symbols in a node's AST. This is done by recursively traversing the node's child nodes and checking if the child nodes are associated with tainted symbols. As illustrated in Algorithm 2, PolyTaint considers that a child node is tainted if *(i)* the child node corresponds to a call to the polyglot API with the SecV language identifier (*i.e.*, "*secV*") or *(ii)* the child node is a value that was tainted before. In the remainder of this section, we outline how PolyTaint performs taint tracking for the different intercepted nodes.

2 PolyTaint instrumentation nodes

**Algorithm 2** — Pseudo-code to check for tainted nodes in AST

```
 1: procedure TraverseAST(node)
 2:     if IsTainted(node) then
 3:         return true
 4:     end if
 5:     for Node child in node.GetChildren() do
 6:         if TraverseAST(child) then
 7:             return true
 8:         end if
 9:     end for
10:     return false
11: end procedure
12: procedure IsTainted(node)
13:     String id ← GetIdentifier(node)
14:     if taintMap.Contains(id) then
15:         return taintMap.Get(id) = 1
16:     else
17:         boolean polygotEvalTest ← node is polyglot.Eval call
18:         boolean secVLiteralTest ← node has "secV" literal node as input
19:         return polyglotEvalTest ∧ secVLiteralTest
20:     end if
21: end procedure
```

**Variable write.** This is a statement that assigns a value to a program variable. For example the JS statement: `var x = y + Polyglot.eval("secV", "sInt(4)")` inside a function's scope is considered a local variable assignment. Similarly, in our illustrative example, statements like `m = polyglot.eval(language="secV", string="sDouble(0.0)")`, `c = polyglot.eval(language="secV", string="sDouble(0.0)")`, and `Y_pred = m * X + c` are variable assignments.

The Truffle instrumentation API provides a generic standard tag operation `StandardTags.WriteVariableTag` [149] to identify nodes corresponding to variable write statements in all Truffle languages. At runtime, PolyTaint wraps every variable write node and creates a corresponding *variable write event node*. The Truffle API makes it possible to obtain a node object descriptor [145] which provides the unique name of the program variable being written to, as well as the corresponding AST node. In the `onReturnValue` callback of the *variable write event node*, PolyTaint leverages node object descriptors to obtain the exact names of the target program variables being written to. For example, for the 3 aforementioned variable write statements, the variable names are respectively `m`, `c` and `Y_pred`.

PolyTaint uses the variable names to construct the unique String identifiers for these variables. Still in the `onReturnValue` callback, PolyTaint checks the *taint map* for the presence of `m`, `c` and `Y_pred`. If they are present, the `onReturnValue` callback returns. Otherwise, PolyTaint leverages the Truffle API to obtain the *right-hand side (rhs)* expression node associated with the variable write statements. Algorithm 2 traverses the child nodes of the *rhs* expressions to check for the presence of any tainted nodes.

As explained previously, the presence of child nodes corresponding to the `polyglot.eval` call as well as the `"secV"` string literal as one of its input arguments tests positive for a *taint source*. This will mark variables `m` and `c` as tainted in the *taint map*, *i.e.*, taint labels of 1 for their identifiers. Similarly, `Y_pred` will be marked as tainted due to the presence of tainted variables `m` and `c` in the *rhs* expression involved in the assignment of `Y_pred`. If a tainted variable is written in a function, this function is also tagged as tainted in the *taint map*. For example, the variable write node corresponding to the statement `Y_pred = m * X + c` is tainted as a result of the presence of tainted variables `m` and `c` in the statement, therefore the enclosing function *trainModel* is marked as tainted in the *taint map*. PolyTaint calls `getName` on the root node object [155] to obtain the unique name of the enclosing function/method for any instrumented node.

**Variable read.** This is a statement that reads the value of a program variable. Variable read operations could occur in an if/else statement, a for loop, a function call via parameters, *etc*. The Truffle instrumentation API provides a generic standard tag `StandardTags.ReadVariableTag` [149] to identify nodes that correspond to variable read statements in all Truffle languages.

At run time, PolyTaint wraps every variable read node and creates a corresponding *variable read event node*. The event node's `onReturnValue` callback is used to obtain the unique name of the variable being read from the node object descriptor, and the *taint map* is checked for the presence of the variable's identifier. If a tainted variable is read in a function (*e.g.*, `while(i < tainted_variable)`), this function is also tagged as tainted in the *taint map*. If a tainted function argument is read (*e.g.*, reading the value of `Y_pred` in `arraySum(Y - Y_pred)` on line 24), the function node is tracked as a neutral function in the *taint map* (*i.e.*, taint label = 2) if it has not been tagged as tainted. In other words, a taint label of 1 for tainted/secure function is preferred for the function node over a taint value of 2 for neutral. As such, `arraySum` is tagged as a neutral function.

**Object field write.** This operation assigns a value to an object's field or property. At the time of this writing, the Truffle API does not provide standard tags to identify generic object field write nodes but Truffle languages typically provide language-specific tags to identify such nodes. For example, Truffle-JS provides the `WritePropertyTag` [144] to identify object property writes. In some languages like JS, the global scope is an object and global variables are properties of this object. As such, global variable writes in JS are instrumented via the `WritePropertyTag`. Object field writes are instrumented with *object field write event nodes* in a similar fashion to variable writes.

**Object field read.** This operation reads an object's field or property. Similarly to object field writes, since there are no generic standard tags yet in the Truffle API to identify object field read nodes, Truffle languages typically provide language-specific tags like `ReadPropertyTag` [144] in Truffle-JS to identify an object field read node. Object field reads are instrumented with *object field read event nodes* in a similar fashion to variable reads.

**Array element write (resp. read).** This is an operation that writes (resp. reads) a value to an array object, *e.g.*, `array[0] = 4` (resp. `if(array[3] < 0)`). Array element writes (resp. reads) are instrumented in a similar fashion to object field writes (resp. reads), with *array element write* (resp. *read*) *event nodes*.

**Function call.** This is an expression that passes control (and possible arguments) to a function or method in a program. The Truffle instrumentation API provides a generic standard tag `StandardTags.CallTag` [149] to identify language nodes that correspond to guest language functions and methods in all Truffle languages. At runtime, PolyTaint wraps every function call node and creates a corresponding *call event node*. The call event node's `onReturnValue` callback is used to test first for tainted arguments.

Truffle languages usually provide methods to obtain argument nodes during instrumentation, *e.g.*, `getArgumentNodes` provided by Truffle-JS (`JSFunctionCallNode` [143]) and Truffle-Python (`PythonCallNode` [148]). If an argument node for a function corresponds to a tainted symbol, *e.g.*, a tainted variable, this function is tagged as neutral in the *taint map* if it has not been tagged as tainted.

For example, `arraySum` is tagged as neutral by virtue of the presence of the tainted variable `Y_pred` in the argument expression.

PolyTaint maintains a list (*seen list*) that comprises all functions/methods that have been visited during execution at run time. Every function/method in the list is identified as a `PolyTaintFunction` object, which is an instance of `PolyTaintFunction` class. This class defines attributes representing the actual function node (*i.e.*, `Node` object), a list representing the different argument types passed to the function (*e.g.*, `int` for `trainModel`, `Object` for `arraySum`, *etc.*), a `String` representing the return type (*e.g.*, `double` for `arraySum`, `void` for `trainModel`, *etc.*) of the function, and an integer value representing the function's taint label. The `onReturnValue` callback of every event node contains a `VirtualFrame` [142] parameter comprising the actual argument values passed to the instrumented node, *e.g.*, a function call node, and an `Object` parameter representing the actual result received after the node is executed. PolyTaint leverages these two parameters to obtain the corresponding argument types passed (if they exist) to every function call node and return types, respectively. The input and return types are used later on during program partitioning (§5.4.5).

At the end of program instrumentation with PolyTaint, all application functions which have been executed at run time (in the *seen list*), but have not been tagged as trusted/tainted or neutral in the *taint map* are considered untrusted functions. As such, for our illustrative example, after instrumentation, we should have set $T$: `trainModel`, set $N$: `arraySum`, and set $U$: `readXData` and `readYData`. This information is then passed to the program partitioner which builds two programs representing the trusted and untrusted partitions of the instrumented program.

### *4.4.4 Program partitioning*

The aim of the partitioning stage is to separate the original program into two parts: a *trusted* part which executes inside the enclave and comprises functions $T \cup N$, and an *untrusted* part which executes outside the enclave and comprises $U \cup N$. By removing the $U$ functions from the enclave, we reduce the size of the TCB. Furthermore, eliminating $U$ functions also eliminates any functions of the language runtime's system libraries invoked by $U$, which are often very large.

In order to interpret the ASTs of the $N$ and $T$ functions inside the enclave at run time, we have to also embed the corresponding Truffle language interpreter in the enclave. However, Truffle is written in Java, which means that we need a Java runtime inside the enclave to execute the Truffle interpreter. Since embedding a full JVM with its system library inside the enclave would increase the size and the attack surface of the TCB to an unacceptable degree, we choose to base the design of our partitioning tool on native images, covered previously in Chapter 2 (§2.2.2) and Chapter 3 (§3.2). GraalVM's native-image tool makes it possible to include only reachable program elements (*i.e.*, methods, classes, and fields) into the resulting native image, making it suitable for restricted environments like Intel SGX enclaves. Any unused classes in the Truffle interpreters or GraalVM's runtime components (*e.g.*, garbage collector) are pruned out of the native images. The remainder of this section describes our partitioning approach.

#### 4.4.4.1 *Generated functions*

GraalVM AOT only supports compiling Java applications to create native images (*i.e.*, no JS, Python, *etc.*), therefore, we need to find a way to run the partitioned guest application code (implemented in JS, Python, *etc.*) via native images. To achieve this, the *partitioner*[3] leverages Truffle's polyglot API to embed the guest code inside two Java programs: `Trusted.java` for the trusted partition and `Untrusted.java` for the untrusted partition. This is done by creating static Java methods that

3 SecV's program partitioner

correspond to each of the methods seen during taint analysis (*i.e.*, $T$, $N$, and $U$). These static methods simply execute the corresponding ASTs of the guest functions they represent.

For instance, Listing 11 illustrates how the AST of a JS function `multi` can be executed from within a Java application, and how a Java method `hello` can be executed from within a JS program.

```java
import org.graalvm.polyglot.*;
public class Example {
  public static void main(String[] args) {
    Context ctx = Context.newBuilder().allowAllAccess(true).build();
    Value jsMulti = ctx.eval("js", "(function multi(a, b){return a*b;})");
    Value res = jsMulti.execute(6, 7);
    System.out.println(res); // prints 42 in Java scope
    Value javaHello = ctx.asValue(Polyglot.class).getMember("static").getMember("hello");
    String jsStringFunc = "function jsHello(javaHello){javaHello();}jsHello;"
    Value jsFunction = ctx.eval("js", jsStringFunc);
    jsFunction.execute(javaHello); // prints "Hello Java" in js scope
  }
  public static void hello() {
    System.out.println("Hello Java");
  }
}
```

**Listing 11:** Java host and guest JS interaction via polyglot API.

Using the same idea outlined in Listing 11, `Trusted.java` therefore comprises static Java methods for `trainModel`, `arraySum`, `readXData` and `readYData`. The static Java methods for `trainModel` and `arraySum` execute the corresponding AST code, *i.e.*, the actual guest source code for those functions, while the static Java methods for `readXData` and `readYData` (proxy methods) perform `ocall` transitions to execute the real methods in the untrusted partition (see below). The actual source code of a function is obtained via the `getSourceSection` method [155] of the TRUFFLE `Node` class. This is illustrated by Listing 12.

```java
public class Trusted {
    Context ctx = Context.newBuilder().allowAllAccess(true).build();
    public static void trainModel(int iter){
        Value arraySumVal = ctx.asValue(Trusted.class).
            getMember("static").getMember("arraySum");
        ctx.eval("python", "// trainModel code").
            execute(iter, arraySumVal, ...);
   }
  public static double arraySum(Object array){
    double ret = ctx.eval("python", "// arraySum code").
        execute(array).asDouble();
    return ret;
   }
  public static Object readXData(int n){
    ocall_readXData(n);
  }
  ...
}
```

**Listing 12:** Trusted.java after partitioning

Similarly, `Untrusted.java` comprises static Java methods for `trainModel`, `arraySum`, `readXData` and `readYData`. The static Java methods for `readXData`, `readYData` and `arraySum` execute the corresponding AST code for those functions while the static Java method for `trainModel` performs an `ecall` transition to execute the method in the trusted partition. Listing 13 exemplifies this.

```java
public class Untrusted {
  Context ctx = Context.newBuilder().allowAllAccess(true).build();
  public static Object readXData(int n) {
    return ctx.eval("python","// readXData code here").execute(n);
  }
  public static void trainModel(int n) {
    ecall_trainModel(n);
  }
  ...
}
```

**Listing 13:** Untrusted.java after partitioning

#### 4.4.4.2 *Transition between the partitions*

With our program partitioned, we must allow communication between the two partitions. That is, how does a Java method in `Trusted.java` (*i.e.*, the trusted partition) invoke another Java method in `Untrusted.java` (*i.e.*, the untrusted partition)?

To achieve this, we leverage GraalVM *C entry points* [141, 225]. As covered in the previous chapter, these are special Java methods in a native image program which are callable from a regular C/C++ program or another native image. As such, C entry point methods can act as relays between the trusted and untrusted partitions. That is, a proxy function in partition-$x$ can invoke the corresponding real static method (`m`) in the opposite partition-$y$ if there is a *C entry point* method for `m` in partition-$y$.

For the trusted partition (*i.e.*, `Trusted.java`), the program partitioner generates C entry point methods corresponding to all the trusted functions. These entry point methods are the target of `ecalls` from the untrusted code calling a trusted function in the final SGX application. Similarly, for the untrusted partition (*i.e.*, `Untrusted.java`), C entry point methods are generated for the untrusted functions. These entry point methods are the target of `ocalls` from the trusted partition calling an untrusted function.

#### 4.4.4.3 *Serialisation*

Because objects cannot be sent across an enclave boundary [30, 225], any static methods that have non-primitive input or return types (*e.g.*, arrays, strings, *etc.*) to be transferred across the enclave serialise the input or return values into a byte array. The byte array is then marshalled across the enclave boundary in the corresponding `ecall` or `ocall`, and deserialised on the opposite side into a Java `Object`. Serialisation and deserialisation are done using Java's `ObjectOutputStream` and `ObjectInputStream` [146] classes, respectively.

```java
public class Trusted {
  // proxy method for readXData
  public static Object readXData(int n) {
    byte[] bytes = ocall_readXData(n); // invoke ocall
    Object xdata = deserialise(bytes);
    return xdata; // can be accessed in any guest scope
  }
}
public class Untrusted {
  public static Object readXData(int n) {// real readXData method
    Object xdata = read(...);
    byte[] bytes = serialise(xdata);
    return bytes;
  }
}
```

**Listing 14:** Serialising and deserialising data that is transferred across the enclave boundary.

#### 4.4.4.4 *Execution of a function*

Thanks to native image, the trusted and the untrusted partitions execute the (binary) compiled version of the corresponding Truffle guest language interpreter at runtime. At runtime, the guest language interpreter parses the string that represents the guest code, builds the AST, and executes it. When the AST of a function `f` contains a call to another function g, the symbol table of the interpreter may not contain the symbol g. In that case, the Truffle API leverages Java reflection to find a static Java method that corresponds to the symbol g and executes it. As an example, in Listing 11, Java reflection is used to expose the Java function `javaHello` (or the JS function `multi`) to Truffle interpreter when executing `jsHello`.

| Partitioning tool | Supported languages | Frameworks used + LoC | Tool's LoC |
|---|---|---|---|
| Civet | Java | SOOT + Phosphor ($\approx 421K$ LoC) | 6,870 |
| Montsalvat | Java | Javassist ($\approx 38K$ LoC) | 3,500 |
| Glamdring | C | Frama-C + Program slicer ($\approx 90K$ LoC) | 5,000 |
| SecV | JS, Python, R, *etc.* | Truffle ($\approx 276K$ LoC) | 4,880 |

**Table 4.1:** Total lines of code for existing partitioning tools and their underlying frameworks.

### 4.4.5 *Building native images and the SGX program*

The aim of the native image generator is to AOT compile `Trusted.java` and `Untrusted.java` into relocatable object files (`trusted.o` and `untrusted.o` respectively). When building a native image, GraalVM makes it possible to provide which guest language implementations (*i.e.*, the Truffle interpreters) should be made available in the resulting native image. In the case of SecV, all guest languages involved in the partitioned program are provided as inputs to the *native image generator*. This tool performs static analysis [211] to determine the reachable Java elements, *i.e.*, the classes, methods, and objects from the Java program that is being compiled and the Truffle language implementations that are required to run the corresponding Java programs. These reachable components are then AOT compiled into native images: `trusted.o` and `untrusted.o`.

It is worth noting that the native image builder does not AOT compile guest language code (*e.g.*, JS, Python, *etc.*) that is embedded in the Java programs. Indeed, the guest language code will be interpreted or JIT compiled by GraalVM at run time. The `ecall` and `ocall` definitions are compiled into object files and linked with `trusted.o` and `untrusted.o`, as well as SGX C/C++ library code, to create the final Intel SGX application. A small shim library is included inside the enclave which seamlessly relays unsupported system calls (*e.g.*, `read`, `write`, *etc.*) to the untrusted runtime via `ocalls`, which perform the real system calls and return the results back to the enclave.

## 4.5 EVALUATION

The experimental evaluation of SecV seeks to answer the following research questions:

**Q1**: What is the implementation effort for a multi-language tool (*i.e.*, SecV) as compared to language-specific tools? (§4.5.2)
**Q2**: What is the cost of injecting SecV nodes into a program AST? (§4.5.3)
**Q3**: What is the cost of taint tracking with PolyTaint? (§4.5.4)
**Q4**: How does partitioning affect application performance? (§4.5.5)
**Q5**: What degree of TCB reduction is achieved by SecV? (§5.6.4)

### 4.5.1 *Experimental setup*

Our evaluation is conducted on a server equipped with a quad-core Intel Xeon E3-1270 CPU clocked at 3.80 GHz, and 64 GB of DRAM. The processor has 32 KB L1 instruction and data caches, a 256 KB L2 cache, and a 8 MB L3 cache. The server runs Ubuntu 18.04.1 LTS 64 bit and Linux 4.15.0-142. We run the Intel SGX platform software with version 2.16 of the SDK and and version 2.14 of the driver. The EPC size on this server is 128 MB, of which 93.5 MB is usable by enclaves. The enclaves have maximum heap sizes of 8 GB and stack sizes of 8 MB. All native images are built with a maximum heap size of 4 GB. We use GraalVM version 22.1.0. All reported measurements are averaged over 5 runs.

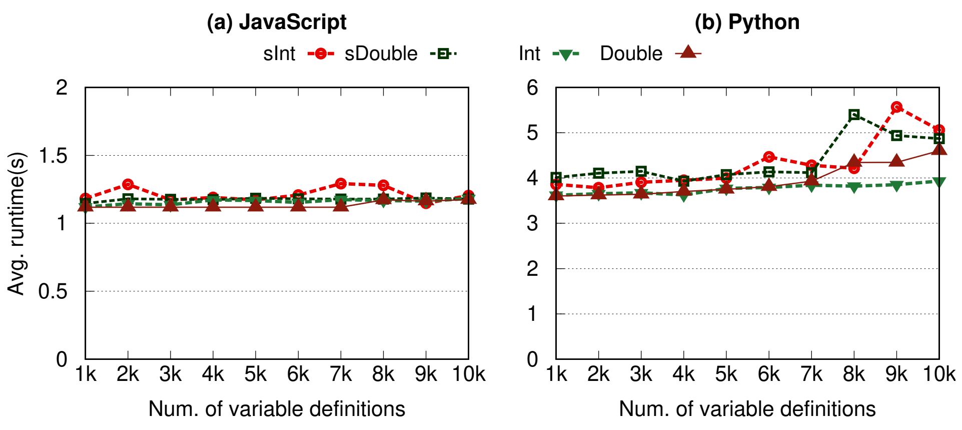


**Figure 4.6:** Cost of using secure values in JS and Python programs.

### *4.5.2 Implementation efforts*

*Q1: what is the implementation effort for a multi-language tool as compared to language-specific tools?*

Table 4.1 reports the total number of lines of code for various enclave code partitioning tools, as well as the LoC for the underlying frameworks used by these tools. We observe that SECV's code base is ≈ 1.41× and ≈ 1.02× smaller as compared to the full code bases of Civet and Glamdring, respectively. We argue that the SECV approach is better in terms of code simplicity (leverages a single self-contained framework) and efficiency (achieves the desired functionality with fewer lines of code). Furthermore, SECV provides a single extensible multi-language system in relatively fewer LoC, contrary to the other tools which are language specific.

### *4.5.3 Overhead of SecV nodes*

*Q2: what is the cost of injecting* SECV *nodes into a program's AST?*

The goal of this experiment is to measure the overhead caused by the use of secure nodes in a polyglot program. In this regard, we generate synthetic programs in JS and Python that comprise functions with varying numbers of variables that receive secure values (*e.g.*, secure `int`, `double` values) via the polyglot API. We compare the run time of the functions with a similar setup where the functions define regular values, *i.e.*, no injection of SECV nodes via the polyglot API. Figure 4.6 shows the results obtained for different value types used.

*Observation.* For JS programs, using `sInt` variables is ≈ 1.05× slower on average as compared to using regular integer variables without the polyglot API. Moreover, using `sDouble` variables is ≈ 1.04× slower on average as compared to using regular `double` type variables without the polyglot API. For Python programs, using `sInt` variables is ≈ 1.14× slower on average as compared to using regular `int` type variables without the polyglot API. Furthermore, using `sDouble` variables in Python is ≈ 1.11× slower on average as compared to using regular `double` type variables.

*Discussion.* For both JS and Python, using SECV nodes involves calls to the TRUFFLE polyglot API, as opposed to defining regular program variables for which there is no intermediate API involved. This explains the additional cost when introducing secure values in the program. In practice, programs will typically define few secure variables leading to a small overhead relative to the total runtime cost of the full program

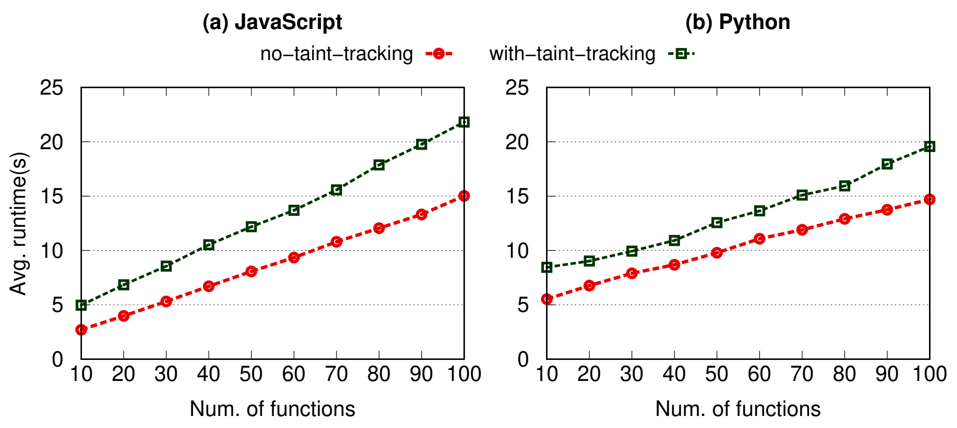


**Figure 4.7:** Cost of taint tracking with PolyTaint for JS and Python.

**Take-away 1**

The cost of injecting secure nodes into a program's AST is small.

### *4.5.4 Overhead of taint tracking*

*Q3: what is the cost of taint tracking with* PolyTaint*?*

In this experiment, we aim to measure the performance of taint tracking with PolyTaint. To that end, we generate synthetic polyglot programs in JS and Python. The synthetic programs consist of a varying number of functions that all perform a bubble sort on a SecV array of 1,000 randomly generated numbers. We chose a bubble sort algorithm here because it performs many variable read/write operations, which trigger the creation of taint tracking instrumentation nodes in PolyTaint. The programs run on GraalVM using their corresponding Truffle interpreters, with or without PolyTaint taint tracking enabled. Figure 4.7 shows the comparative performance of both setups.

*Observation.* The experimental results show that the JS program running with PolyTaint taint tracking enabled is ≈ 1.56× slower when compared to the JS program running without taint tracking enabled. Similarly, for the Python program running with PolyTaint taint tracking enabled, it is ≈ 1.31× slower on average when compared to the variant without taint tracking enabled.

*Discussion.* For both JS and Python programs, PolyTaint introduces more nodes (*i.e.*, wrappers and execution event nodes) to their ASTs during instrumentation. As seen in §4.4.3, those nodes perform operations to track tainted variables, which explains the performance drop (*i.e.*, longer run time) when taint tracking is enabled, as compared to the variants where the programs run without any taint tracking.

**Take-away 2**

Taint tracking via AST instrumentation introduces overhead at run time. This overhead is due to the additional operations performed by instrumentation nodes introduced in the program's AST.

### *4.5.5 Effect of partitioning on performance*

*Q4: how does partitioning affect application performance?*

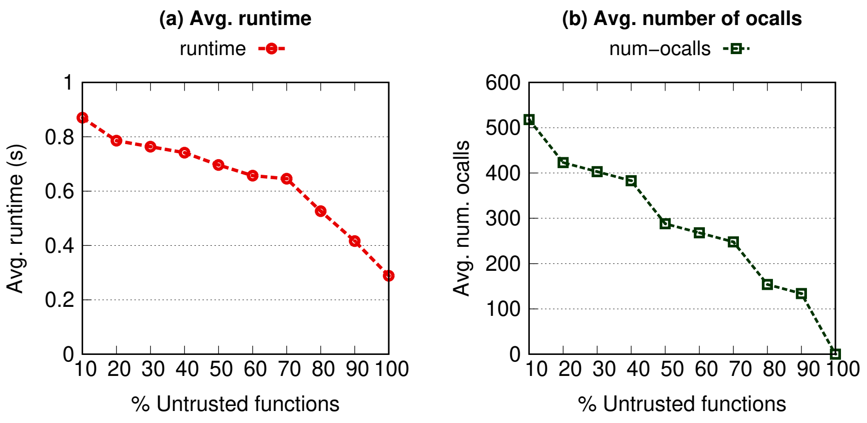


**Figure 4.8:** Effect of program partitioning on a generic synthetic program.

To study the impact of partitioning on application performance with SecV, we first use a synthetic polyglot benchmark written in JS. We then partition two real-world applications that implement the *PageRank* algorithm [3] and the *linear regression* ML algorithm [216].

**Synthetic benchmark.** Our synthetic benchmark is a JS program that comprises 100 functions that each perform a bubble sort on a local array variable of size 100. We leverage secure values (secure array or regular array) in a varying percentage of the functions, analyse the polyglot programs with PolyTaint and partition these programs for enclaves. The purpose of our synthetic benchmark is to highlight the effect of partitioning on a generic program. Figure 4.8 shows the results.

We observe that as more functions are partitioned out of the enclave, the overall performance of the program improves. This is explained firstly by the reduced number of `libc`-related `ocalls` performed by the embedded GraalVM runtime components (language interpreters, GC, *etc.*), and secondly by the fact that we have less overhead due to enclave data encryption-decryption operations to and from the EPC by the MEE [225].

**Take-away 3**

Enclave performance improves as more computations are delegated to the untrusted side. This is due to less expensive enclave context switches, *e.g.*, via ocalls, as well as less expensive enclave-related cryptographic operations.

We have observed the effect of partitioning on a generic synthetic program. We now leverage SecV to partition real-world applications. For these applications, we evaluate 3 modes for running the programs: entirely inside the enclave (`no-part`), as a normal polyglot native image without SGX (`native`), and as a partitioned native image using SecV (`part`). The partitioning scheme adopted for each application is not necessarily realistic, but aims to highlight and explain the performance improvement obtained after partitioning.

**PageRank.** PageRank [3] is a popular algorithm that is used in graph processing frameworks [93] to weigh the relative importance of nodes (*i.e.*, node ranks) in a directed graph. One could envision a scenario where the node ranks are to be secured within an enclave. We leverage SecV to partition a PageRank program with the node rank data structure (a secure array) tagged as a secure variable. The PageRank program comprises a function to generate a directed graph and obtain the corresponding adjacency list (graph pre-processing), and other functions which use the adjacency list to perform the PageRank algorithm. The graphs are generated using the RMAT algorithm [17], and all graphs with

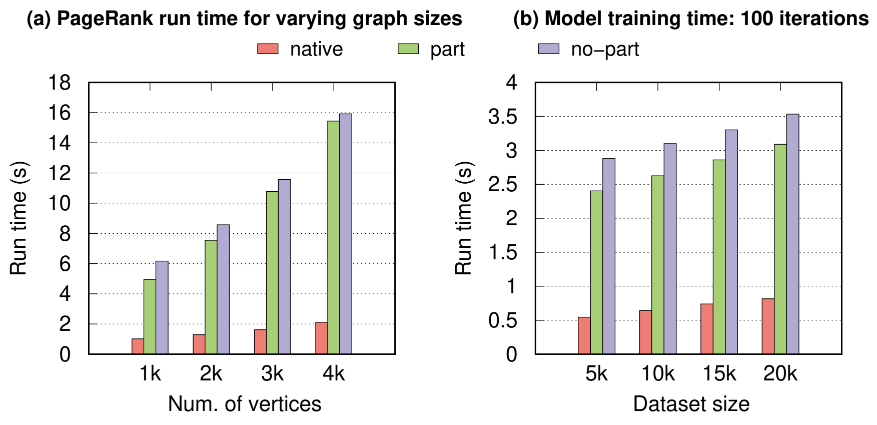


**Figure 4.9:** Results for partitioning the PageRank and Linear Regression programs.

$n$ vertices have $2 \cdot n$ edges. Figure 4.9 (a) shows the results obtained when running partitioned and unpartitioned versions of PageRank with varying graph sizes.

*Observation.* The partitioned version of the PageRank program is $\approx 1.12\times$ faster on average (*i.e.*, about 10.8% performance improvement) as compared to the unpartitioned version. Furthermore, we have on average $\approx 2.53\times$ fewer `ocalls` in the partitioned program as compared to the unpartitioned one. The native version of the program (*i.e.*, no SGX) is $\approx 6.85\times$ and $6.17\times$ faster on average as compared to the unpartitioned and partitioned versions, respectively.

*Discussion.* After partitioning the program, the functions responsible for graph generation and pre-processing are moved to the untrusted partition, while those that perform the PageRank algorithm to obtain node ranks remain in the trusted partition. At runtime, the pre-processed graph (in the form of an adjacency list) is serialised in the untrusted partition and marshalled into the enclave (see §4.4.4.3) via an `ocall`. It is then deserialised to recreate the adjacency list object which is used by the in-enclave PageRank algorithm to compute the final page ranks. The computations outside the enclave runtime do not incur expensive context switches via `ocalls`, or SGX-related cryptographic operations. That explains the performance improvement as well as the reduced number of `ocalls` in the partitioned version as compared to running the full program inside the enclave. This also explains why the native application is fastest: no SGX-related cryptographic operations, no enclave context switches, and less memory restrictions (the enclave has only $\approx 93.5MB$ of memory). However, it is the least secure of all.

**Linear regression.** Securing ML models with TEEs is common [223, 122]. We aim to partition a linear regression program for an enclave runtime. Similar to PageRank, a linear regression program typically consists of functions to read and pre-process (*i.e.*, normalisation and standardisation) the input datasets to be used, and other functions which use the dataset to train the ML model. We leverage SecV to specify secure variables (*i.e.*, `m` and `c` in Listing 8), analyse and partition the linear regression program. Figure 4.9 (b) shows the results when the linear regression model is trained for 100 training iterations for partitioned and unpartitioned versions of the program.

*Observation.* The partitioned version of the linear regression program is $\approx 1.17\times$ faster on average (*i.e.*, about 14.5% performance improvement) as compared to the unpartitioned version. Moreover, we have on average $\approx 1.36\times$ fewer `ocalls` in the partitioned program when compared to the unpartitioned one. The `native` version of the program is $\approx 4.73\times$ and $4.04\times$ faster on average when compared to the unpartitioned and partitioned versions, respectively.

**Table 4.2:** Size in LoC of in-enclave components for SecV and a LibOS-based system.

| Component | ≈ LoC | SecV | LibOS |
|---|---|---|---|
| TruffleJS | 236,340 | ✓ | ✓ |
| GraalPython | 1,231,920 | ✓ | ✓ |
| SecV node generator | 1280 | ✓ | ✗ |
| SubstrateVM | 210,780 | ✓ | ✗ |
| Shim library | 11,600 | ✓ | ✗ |
| JVM | 593,159 | ✗ | ✓ |
| JNI runtime libraries | 423,303 | ✗ | ✓ |
| Gramine | 40,540 | ✗ | ✓ |
| GNU libc 2.19 | 1,008,780 | ✗ | ✓ |
| **Total LoC (JS)** | | 460,000 | 2,302,122 |
| **Total LoC (Python)** | | 1,455,580 | 3,297,702 |

*Discussion.* After program partitioning, the functions for data generation and pre-processing (which do not access `m` and `c`) are partitioned out of the enclave, while the trusted functions that train the model (and hence access the secure values `m` and `c`) are part of the enclave partition. At runtime, dataset pre-processing is done in the untrusted partition; this entails standardising both the `X` and `Y` datasets. The standardised datasets (arrays) are then serialised and marshalled into the enclave runtime via `ocalls`, deserialised inside the enclave, and used to train the linear regression model. By offloading the data generation and pre-processing phases to the untrusted runtime, the enclave is relieved of expensive computations, which leads to an overall improvement in application performance when compared to running the full program inside the enclave. Similar to PageRank, partitioning introduces overhead for data serialisation and deserialisation, but this overhead is offset by the performance gain from performing some computations outside (*i.e.*, no expensive `ocalls`), rather than inside the enclave.

### 4.5.6 TCB analysis

*Q5: what degree of TCB reduction is achieved by* SecV*?*

We conduct a TCB analysis of our system along two dimensions. First, we compare the TCB of SecV to a LibOS-based approach that incorporates the JVM within the enclave. We evaluate the number of lines of code for the in-enclave components in both approaches. SecV leverages GraalVM's SubstrateVM (SVM) [134] to provide runtime components for the AOT compiled native images, and a small shim library to relay unsupported libc calls to the untrusted runtime. The LibOS-based approach relies on Gramine (formerly Graphene-SGX) v1.1 [163] and OpenJDK8 JVM. For both SecV and the LibOS-based system, we employ the same Truffle-based interpreters: TruffleJS and GraalPython for JavaScript and Python respectively. The results are outlined in Table 4.2. We observe that SecV achieves 5× and 2.3× reduction in TCB (in LoC) for JavaScript and Python programs respectively, when compared to a LibOS-based system where the LibOS, JVM, libc, as well as interpreters are included inside the enclave. It is important to note that the actual size (in LoC) of the in-enclave components (*e.g.*, TruffleJS, GraalPython) may be less due to pruning during native image building. Nevertheless, we consider the total number of LoC (*i.e.*, the maximum value) in our evaluation because the actual value is difficult to obtain in practice. Conversely, the full component is included in the enclave with the LibOS-based approach.

Subsequently, for each partitioned application, we evaluate the size of the in-enclave components before and after partitioning with SecV. This further reduces the TCB by excluding untrusted functions from the enclave. In the case of PageRank, the in-enclave application code is reduced from 200 to 106 LoC. For the linear regression program, the in-enclave application code is reduced from 145 to 115 LoC.

**Take-away 4**

SecV achieves up to 5× TCB reduction with respect to a LibOS-based approach. This is attributed mainly to the absence of the full JVM, LibOS, and C standard libraries in the enclave.

## 4.6 LIMITATIONS OF OUR DESIGN

**Limited code coverage.** The present design of SecV is based on a purely dynamic program analysis approach, where programs are run once during the development phase to deduce the set of *trusted* ($T$), *neutral* ($N$), and *untrusted* ($U$) functions, which are then used to create the trusted and untrusted partitions. However, this design provides limited code coverage, potentially leaving security-sensitive code out of the enclave; limited code coverage is a fundamental limitation of most dynamic analysis tools [88]. To address this problem, *static analysis* can be applied, but it has major drawbacks. Firstly, it performs an over-approximation of the code to be tainted (due to polymorphism), which leads to a larger TCB. Secondly, Truffle ASTs exist only at runtime, making static analysis infeasible. Ultimately, a mixture of both static and dynamic taint analysis techniques could be a reasonable compromise. We leave this as future work.

**Application termination.** During analysis, different program runs (with varying inputs) could result in different execution paths being taken at runtime. This could in turn result in different (and possibly conflicting) sets for $T$, $N$, and $U$. From a security perspective, such a scenario may lead to secure values being leaked to the untrusted partition at runtime. In our prototype implementation, we avoid this problem by checking if a value is secure just before serialising it in in-enclave proxies. Thanks to this check, in the worst case, the application will terminate with an error, but will not let a secure value escape the enclave. Running multiple tests or using symbolic execution could decrease the probability of terminating the program. Similarly, we leave this implementation as future work.

## 4.7 RELATED WORK

We classify related work into 3 categories: *(i)* tools that run full, unmodified applications inside enclaves, *(ii)* taint tracking tools, and *(iii)* code partitioning tools.

**Running full applications inside enclaves.** Various tools like SCONE [7], Graphene-SGX [195], TWINE [117], and SGX-LKL [160] propose solutions to run entire legacy applications inside enclaves. Their approach severely increases the size of the TCB, at the risk of added security vulnerabilities. SecV provides a generic solution to partition applications for enclaves while trying to keep the TCB as small as possible.

**Taint tracking tools.** Several tools exist for taint tracking. Phosphor [12] is a dynamic taint tracking tool for Java programs, while Dytan [24] performs taint tracking in x86 binaries. TAJ [191] provides efficient static taint analysis for Java applications. [180] provide a tool to extract taint specifications for JavaScript libraries. However, these tools are language-specific (*i.e.*, only for Java, only for x86 assembly code, JS, *etc.*) and cannot be readily leveraged to analyse code in different languages. SecV bridges this gap with a multi-language taint tracking approach. TruffleTaint [92] leverages the Truffle framework to provide a language-agnostic platform to build dynamic taint analysis applications. While we leverage very similar instrumentation techniques, we provide a novel way to specify sensitive values through the introduction of secure AST nodes, and leverage these to partition code for the enclave runtime in a language-agnostic fashion.

**Language specific partitioning tools.** Several tools have been proposed to partition code written in specific languages for enclaves. Glamdring [104] provides a technique to automatically partition C applications, while Montsalvat [225], Civet [197], and Uranus [79] propose solutions to partition Java applications for Intel SGX enclaves. Similarly, these partitioning tools are language-specific. SecV provides a multi-language technique to partition programs for enclaves.

## 4.8 SUMMARY

This chapter introduced SecV, a multi-language approach to analyse and partition programs for Intel SGX enclaves. SecV provides generic *secure nodes* that serve as multi-language annotations for sensitive data. SecV provides a dynamic taint tracking tool, PolyTaint, which tracks the flow of sensitive data from *secure nodes* in a program at run time, and partitions the program into trusted and untrusted parts which are executed in and out of the secure enclave respectively. The experimental evaluation of SecV shows TCB reduction can be achieved in a program without any performance degradation.

CHAPTER 5

# Secure Peripheral I/O with TEEs

The tools proposed so far focused on enhancing security in server-end environments. This chapter, however, aims to tackle security and privacy issues in client-end internet of things (IoT) setups, where vast amounts of security- and privacy-sensitive data are generated. While TEE technologies like Arm TrustZone have the potential to mitigate these issues, the development of high-level frameworks that leverage these technologies is still at an early phase. We aim to bridge this gap by designing and implementing a holistic and generic TEE-based solution for safeguarding peripheral data in IoT setups. We provide a generic partitioning blueprint to assist developers in adapting peripheral device drivers for TEEs.

The chapter is organised as follows:

## 5.1 INTRODUCTION

The *Internet of Things* (IoT) [8] concept has gained tremendous popularity in recent years. In broad terms, IoT refers to the network of smart devices capable of collecting, processing, and exchanging data over the internet. IoT has vast applications across various domains, including smart homes, healthcare, agriculture, industrial automation, and enables a wide range of possibilities, such as remotely controlling devices, monitoring environmental conditions, improving efficiency and enhancing safety.

The nature of interconnected devices and the vast amounts of data generated [4] has led to several security and privacy concerns. For instance, in smart homes, large amounts of sensitive data are constantly generated through sensors and hardware peripheral devices, *e.g.*, cameras, microphones. This data is usually transmitted to untrusted cloud services and third-party providers, oftentimes with little regard to the confidentiality of the shared data. The involvement of multiple parties introduces privacy issues, as the data could be used for purposes beyond the user's knowledge or control. For example, in July 2019, more than 1000 Google Assistant recordings were involuntarily leaked [55, 27], with part of these recordings activated accidentally by users. Furthermore, privileged software like the underlying operating system (OS) or hypervisor can be compromised [94, 200], potentially leading to data breaches or unauthorised access to sensitive data.

TEE technologies have the capability to mitigate these security threats. Contrary to cloud-based systems which employ TEEs like SGX, TDX, and SEV, IoT systems are typically located on the edge, and leverage TEEs like Arm TrustZone. Although TrustZone-based frameworks like OP-TEE [192] have been leveraged to secure both user and kernel level applications in IoT setups, a holistic solution for safeguarding peripheral data remains absent.

This work aims to fill this gap by proposing FORTRESS, a robust and comprehensive framework to enhance security and privacy in IoT infrastructures. Our approach involves restricting peripheral I/O memory to a secure kernel space TEE, providing access only to a small part of peripheral driver code while isolating peripheral data from an untrusted OS or hypervisor. The data is then securely transferred to a user space TEE, where obfuscation or data sanitisation techniques can be applied to encrypt or remove sensitive information before it is transmitted to an untrusted cloud environment. As outlined in §5.7, the practical applications of this security framework extend to diverse contexts such as smart homes, medical IoT, retail IoT, amongst many others. In summary, the contributions of this work are as follows:

- A design to secure IoT peripheral data using Arm TrustZone.
- A generic approach to partition peripheral drivers for Arm TrustZone.
- A proof-of-concept implementation of our design using $\mathtt{I^2S}$ (inter-ic sound) based peripherals on ARMv8-A, demonstrating the feasibility of the proposed approach.
- A comprehensive evaluation of our system, which shows that privacy is achieved at a reasonable cost.

## 5.2 BACKGROUND

This work relies on Arm TrustZone, the predominant TEE technology in low-power devices at the edge, commonly found in IoT setups, *e.g.*, home automation systems, smart sensors, *etc*. TrustZone essentially divides the processor into a secure world and a normal world, dedicated to executing sensitive and non-sensitive operations, respectively. The transitions between these two worlds are managed by a specialized software known as the secure monitor, via secure monitor calls (SMCs). We provide a comprehensive overview of TrustZone's architecture in Chapter 2 (§2.1.3). To simplify the

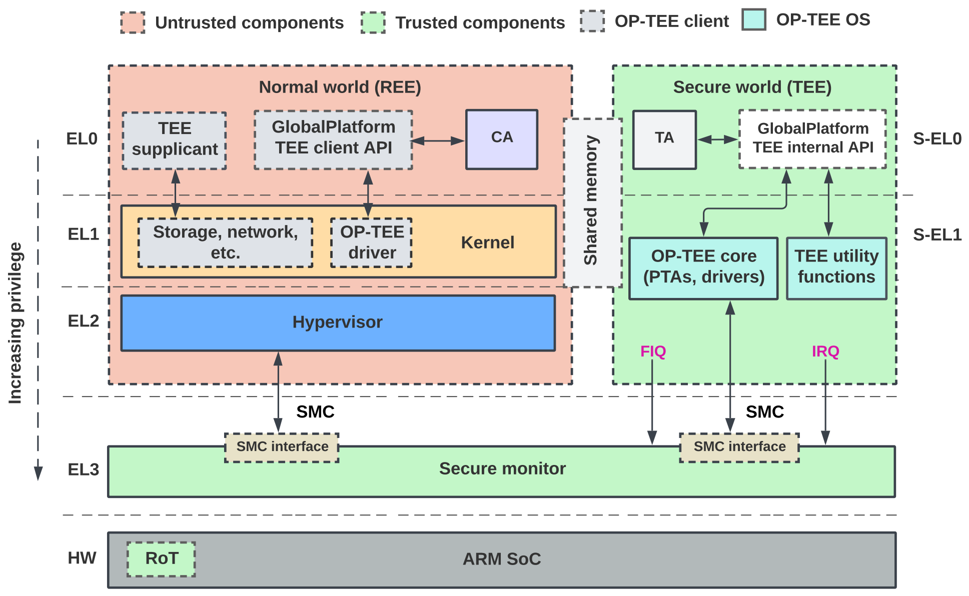


**Figure 5.1:** OP-TEE architecture.

development of secure applications based on TEE technologies like TrustZone, the tool *open portable TEE* (OP-TEE) was developed.

### 5.2.1 *Open Portable TEE (OP-TEE)*

OP-TEE [192] is a software framework that implements a trusted execution environment for TrustZone. OP-TEE is compliant with the GlobalPlatform TEE Internal/Client API specification v1.0 [51], and consists of three main components: *OP-TEE client*, *OP-TEE OS*, and *OP-TEE Linux driver*, as illustrated in Figure 5.1. OP-TEE client offers an API that allows applications in the normal world, also known as *client applications* (CAs), to interact with applications running in the secure world, known as *trusted applications* (TAs). Specifically, the CAs operate within a *rich execution environment* (REE), *i.e.*, in the regular OS at the EL0 level, while TAs execute in the secure world inside the TEE, at the S-EL0 level. OP-TEE Linux driver is a kernel-space component operating in the normal world OS (at EL1) which facilitates transitions between normal and secure worlds. In the presence of a hypervisor, the SMC from a guest kernel traps into the hypervisor and the latter performs the SMC on behalf of the former. OP-TEE OS is an OS designed to run in the secure world (at S-EL1). It provides generic OS-level functions like interrupt handling, thread handling, crypto services, and shared memory. OP-TEE OS implements the GlobalPlatform TEE Internal Core API, and provides an interface called a *pseudo trusted application* (PTA) which TAs can use to communicate with OP-TEE OS.

### 5.2.2 *MMIO and DMA*

Contemporary computer systems and IoT devices comprise peripheral devices, *e.g.*, keyboard, mouse, camera, microphone, fingerprint sensors, aimed to provide I/O capabilities. These peripheral devices generate vast amounts of data that could be sensitive. Dedicated system software called *device drivers* enable the OS kernel to interact with these peripherals, and hence the corresponding data. As illustrated in Figure 5.2, the job of a typical device driver is, for the most part, reading or writing I/O memory [29]. This is commonly achieved through mechanisms like MMIO and DMA, which we detail next.

**MMIO.** It is a method of performing I/O between the CPU and a peripheral device by mapping

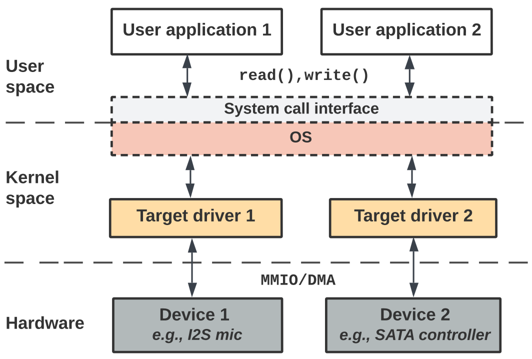


**Figure 5.2:** Device drivers.

```
tegra_i2s1: i2s@2901000 {
    compatible = "nvidia,tegra194-i2s","nvidia,tegra210-i2s";
    reg = <0x2901000 0x100>;
    clocks = <&bpmp TEGRA194_CLK_I2S1>,<&bpmp TEGRA194_CLK_I2S1_SYNC_INPUT>;
    clock-names = "i2s", "sync_input";
    ...
    sound-name-prefix = "I2S1";
    status = "okay";
 };
```

**Figure 5.3:** Device tree node for an I$^2$S interface on Tegra 194 SoC.

the peripheral's *I/O registers* into the CPU's address space. As such, processor `load`/`store` instructions on mapped addresses translate to load/store operations on the corresponding I/O registers in physical memory. The Linux kernel on 64-bit ARM platforms provides MMIO access primitives like `ioreadX`/`iowriteX` which read and write `X` bits of data from and to an I/O register, respectively. For example, `ioread8(reg)` and `iowrite8(reg,val)` read and write 8 bits of data from and to a memory-mapped register, respectively.

**DMA.** It is a mechanism used by peripheral devices to transfer I/O data to and from main memory bypassing the processor [29]. A specialised hardware component, *i.e.*, the *DMA controller*, coordinates the DMA data transfer, and generates an interrupt request to the CPU upon DMA completion.

MMIO or DMA memory for a particular peripheral device is usually associated with a *base address* and *size*, which indicate respectively the base address in physical memory, and the size of the contiguous portion of memory used by the data transfer mechanism. The peripheral's specifications provides information regarding base addresses, sizes, and the offsets for all useful registers and DMA regions. The details regarding all hardware devices on a system are described in a data structure called the *device tree* (DT). The DT contains nodes describing all the hardware on a system, including CPUs, memory, clocks, and peripheral devices like I$^2$S, Ethernet cards, *etc*. For example, Figure 5.3 represents a node in the DT of an Nvidia Jetson AGX Xavier kit describing an I$^2$S peripheral's properties, such as the base address in memory (`0x2901000`) and size (`0x100`). The DT is read by the kernel at boot time.

As outlined in Chapter 2 (§2.1.3), TrustZone provides an address space controller, TrustZone address space controller (TZASC), which can be leveraged to restrict memory regions (*e.g.*, those used by MMIO or DMA) in the trusted world.

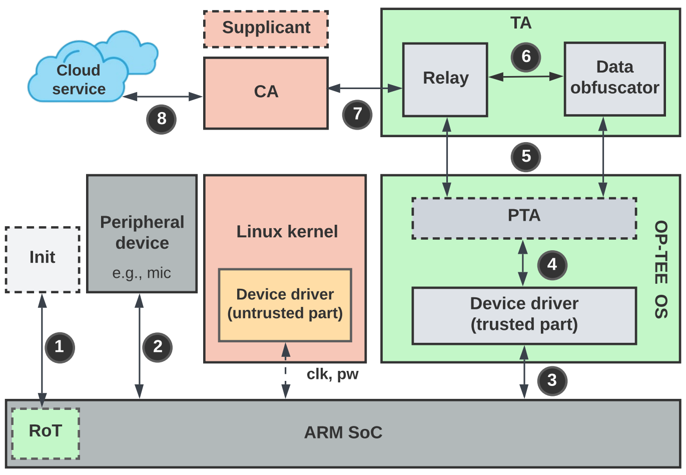


**Figure 5.4:** FORTRESS workflow.

## 5.3 THREAT MODEL

FORTRESS has four main security goals, aimed to guarantee the confidentiality and integrity of sensitive data (*e.g.*, images, voice recordings) originating from IoT peripherals before its transmission to the cloud:

**G1:** Protect the confidentiality of peripheral data from the underlying OS or hypervisor.
**G2:** Ensure a trusted path between kernel space where the data is originally obtained to secure user space where data processing (*e.g.*, encoding, decoding, filtering) occurs.
**G3:** Ensure the confidentiality (and integrity) of sensitive peripheral data transferred to the cloud, or complete removal of sensitive data from the data stream.
**G4:** Minimise the trusted computing base and hence the potential attack surface.

**Hardware.** We assume that the underlying hardware itself is not malicious and cannot be tampered with by the adversary. FORTRESS shields existing peripherical devices connected to the SoC, and does not consider the connection of new components, *i.e.*, the adversary's goal is to access sensitive data of already connected peripherals, typically via MMIO or DMA. Denial-of-service attacks (*e.g.*, maliciously shutting down the system) as well as side-channel vulnerabilities [107] are considered out-of-scope.

**Software.** The on-die boot ROM and intermediate firmware, as well as OP-TEE are considered trusted components, which permit establishment of a *chain-of-trust* during a secure boot process. We assume the peripheral's hardware components information, *e.g.*, MMIO or DMA physical addresses, is encapsulated in a device tree file that is signed and verified while establishing the chain-of-trust, preventing a potential attacker from misconfiguring the device. All other software on the system is considered untrusted.

**Third-party.** All entities outside the hardware perimeter of the SoC, such as the cloud providers in the IoT setup (*e.g.*, AWS IoT [9]) to which potentially sensitive peripheral data is transmitted, are considered untrusted.

## 5.4 FORTRESS ARCHITECTURE

### 5.4.1 *Overview*

In this section, we present the architecture and design of FORTRESS, and show how its components interact to achieve the security goals outlined in §6.5.

The workflow of FORTRESS is summarised in Figure 6.5. At system initialisation, the secure boot process authenticates system boot components, *e.g.*, OP-TEE OS image, DT files. TZASC is then used to configure peripheral MMIO and DMA memory regions to be accessible to the secure world only (❶). After system setup, a peripheral device, *e.g.*, microphone or camera which constitutes a smart home/IoT setup, generates potentially sensitive data, *i.e.*, speech or visual data (❷). In a regular setup, the device driver software is part of the untrusted OS, thus leaking sensitive data. In FORTRESS, the driver is partitioned into trusted and untrusted parts, which respectively execute in the secure and normal world. The trusted part is embedded in the OP-TEE OS kernel, and has exclusive access to the generated peripheral data, which is read into secure I/O buffers (❸). The sensitive data is securely processed by the trusted driver, after which it is transferred to a trusted application via the PTA interface (❹–❺). The TA comprises a data obfuscator that acts as a firewall, encrypting or filtering out any sensitive information (❻). Converting human speech into a finite set of voice commands is a typical example of stripping potentially sensitive information. Finally, the reviewed data is sent to an untrusted cloud service via a relay module in the TA (❼–❽). The relay module leverages a Linux user-space daemon called the *TEE supplicant* to provide OS-level services such as network communication or storage.

We create a proof-of-concept implementation of this approach with secure sound processing via $I^2S$, a serial communication protocol commonly used to transmit audio data between integrated circuits (ICs). Figure 5.5 provides an overview of the $I^2S$ protocol: One IC, *i.e.*, SoC (master) controls data transfer by manipulating two clock lines: the *left-right clock* to specify the audio channel (`L=0;R=1`), and the *bit clock* to specify if data can be sent (`1`) or not (`0`). The second IC, *i.e.*, $I^2S$ microphone (slave) sends data bits (via MMIO or DMA) for the corresponding channel each time the bit clock is set.

### 5.4.2 *System initialisation and peripheral memory isolation*

**Secure boot.** To ensure the authenticity and integrity of trusted system components, Arm TrustZone supports *trusted board boot* (TBB) [6]. The latter establishes a chain-of-trust consisting of several stages of integrity verifications, starting from the boot ROM and extending through various bootloader stages, the Arm trusted firmware (containing the secure monitor), and OP-TEE OS components. During the deployment phase, critical components of the TCB, such as OP-TEE OS, device tree files, and TAs undergo cryptographic signing (using SHA-256) with a private key. The secure boot process uses the corresponding public key to verify the signatures of these components. By verifying the integrity of OP-TEE OS, this in turn ensures the integrity of FORTRESS's trusted peripheral driver, as well as device tree files containing peripheral I/O registers.

**Memory isolation.** To safeguard peripheral data from unauthorised access by untrusted components like the OS and hypervisor, it is necessary to confine the memory regions associated with peripheral data to the secure world. In the context of our work, we collectively refer to these memory regions as the *secure I/O region* of the peripheral. Since most IoT hardware peripherals transfer data using MMIO or DMA mechanisms, FORTRESS configures peripheral MMIO and DMA memory ranges as the secure I/O region. The specific MMIO registers and DMA regions used by the peripheral are typically defined

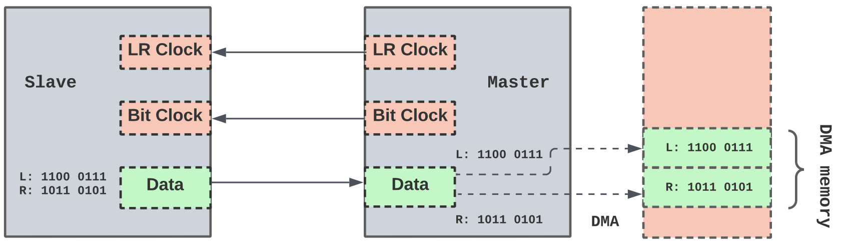


**Figure 5.5:** Overview of I$^2$S protocol.

in the device's DT node. Our solution offers two methods for specifying these secure I/O regions. The first is static and involves hard-coding the physical memory addresses of the MMIO and DMA base registers (along with their sizes) directly into the driver's source code. The second approach is dynamic, where the information is read from the DT node in the corresponding device tree file.

During system initialisation, the trusted firmware utilises TZASC to configure DRAM in such a way that the peripheral's secure I/O region (*i.e.*, MMIO and DMA memory ranges) is exclusively accessible to the secure world. This configuration effectively prevents any access to the peripheral's data from the untrusted OS or hypervisor, thereby satisfying the security requirement in **G1**. To complete the initialisation of the peripheral's memory, the trusted driver's initialisation code invokes OP-TEE OS's `core_mmu_add_mapping` routine, which maps the secure MMIO regions into kernel-space,[1] ensuring that the secure driver can effectively interact with the peripheral.

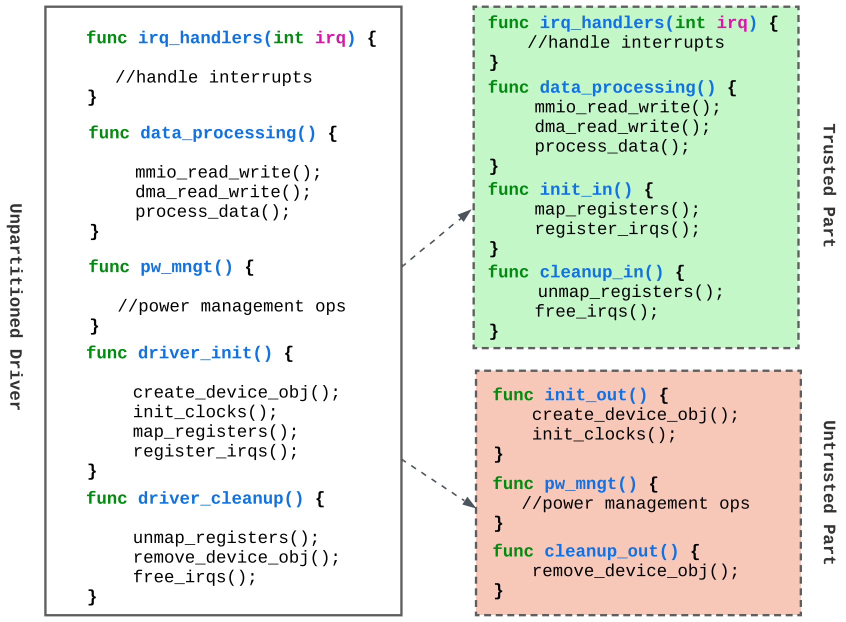
### 5.4.3 Secure kernel to user-space communication

After the trusted driver reads peripheral data via MMIO or DMA into its I/O buffers, it needs to transfer the data to a secure user-space TA for further processing operations, *e.g.*, encoding, encryption, or filtering. To establish this secure connection between the OP-TEE driver and the TA, we leverage an OP-TEE PTA. The latter acts as a bridge, offering an interface that connects the TA to secure driver space. The TA utilises GlobalPlatform Internal API to safely retrieve peripheral data from the driver's memory space via a PTA invocation. This operation requires a context switch from S-EL0 to S-EL1, and can be summarised in four steps: *(1)* checking access rights of the TA's destination buffers in the invocation, *(2)* copying necessary parameters (*e.g.*, the PTA/driver function identifier) from TA memory to OP-TEE OS memory space, *(3)* issuing a system call to OP-TEE OS to invoke the corresponding PTA/driver routine, and *(4)* copying results from OP-TEE OS memory into the TA's destination buffer. The creation of this secure channel from OP-TEE OS to the TA satisfies **G2**.

### 5.4.4 Data obfuscation

*Data obfuscation* is the process of transforming sensitive information to safeguard it from unauthorised access. Common data obfuscation techniques include encryption, randomisation, anonymisation *etc*. By integrating a data obfuscation module in FORTRESS, sensitive data originating from an IoT peripheral can be protected before its transmission to an untrusted party in the cloud or the host OS, hence satisfying **G3**. The generic architecture proposed by FORTRESS offers the flexibility to incorporate diverse obfuscation techniques depending on the nature of the data. Our implementation leverages techniques provided by OP-TEE's cryptography API which is based on LibTomCrypt [101], a popular open-source cryptographic library. The latter provides various symmetric cryptographic algorithms, including AES-GCM, which can be leveraged to encrypt and guarantee the integrity of sensitive data. Public-key encryption techniques, although viable, were

1 Mapping I$^2$S MMIO registers in the secure world



```
func irq_handlers(int irq) {

   //handle interrupts
}

func data_processing() {

     mmio_read_write();
     dma_read_write();
     process_data();
}

func pw_mngt() {

   //power management ops
}
func driver_init() {

     create_device_obj();
     init_clocks();
     map_registers();
     register_irqs();
}
func driver_cleanup() {

     unmap_registers();
     remove_device_obj();
     free_irqs();
}
```

```
func irq_handlers(int irq) {
    //handle interrupts
}
func data_processing() {
     mmio_read_write();
     dma_read_write();
     process_data();
}
func init_in() {
     map_registers();
     register_irqs();
}
func cleanup_in() {
     unmap_registers();
     free_irqs();
}
```

```
func init_out() {
    create_device_obj();
    init_clocks();
}
func pw_mngt() {
   //power management ops
}
func cleanup_out() {
    remove_device_obj();
}
```

**Figure 5.6:** FORTRESS driver partitioning.

not selected for our design due to their performance overhead when compared to their symmetric counterpart. Additional data obfuscation methods, including *data conversion* and *data filtering*, serve to reduce the sensitivity of the data or completely strip sensitive elements, respectively. Data conversion can be particularly effective for human voice recordings, converting them into voice commands upon recognition of specific voice patterns. Data filtering, on the other hand, can entirely remove unrecognised sentences. The primary advantage of data obfuscation is that it effectively restricts the distribution of sensitive data in uncontrolled environments, such as the REE and the cloud service providers.

### *5.4.5 Driver partitioning*

A key aspect of TEE development is decreasing the trusted computing base. As previously discussed, TCB reduction is usually done via *code partitioning*. For an IoT-based system like FORTRESS, this involves identifying portions of the code that manipulate sensitive peripheral data and isolating only these within the TEE; the non-sensitive part can be kept out of the TEE. OP-TEE provides some examples of how to build secure peripheral drivers but lacks a systematic approach for partitioning them into trusted and untrusted components.

The aim of this section is to provide a high-level approach to guide developers in partitioning drivers for securing peripheral data in FORTRESS. We illustrate our approach by considering a simple $I^2S$ peripheral driver to be partitioned.

In FORTRESS, we perform *manual partitioning* of the driver into *trusted* and *untrusted* parts. To determine which part of the driver to secure in the TEE, one must first identify the driver code that interacts with the peripheral's secure I/O region. In the case of $I^2S$, the microphone generates audio data which is transferred to memory via DMA or MMIO. MMIO registers are provided by the SoC used for $I^2S$ control, such as enabling/disabling $I^2S$, or specifying audio data formats. In this case, the trusted driver code integrates the complete logic required for mapping MMIO registers, for performing read and write operations on these MMIO registers and DMA buffers. Additionally, the code manages interrupt handling, such as when a DMA operation terminates. The remaining code, which generally comprises clock and power management operations, is delegated to the untrusted part driver since this code does not access the data being secured.

Based on the overall structure of peripheral drivers as well as the data processing techniques (*i.e.*, MMIO, DMA) previously discussed, we provide in Figure 5.6 a generic blueprint for manually partitioning a driver with FORTRESS, satisfying **G4**. For more complex code bases, techniques such as *static data-analysis* [25, 86] could be leveraged to properly identify which code portions interact with sensitive MMIO or DMA regions of the peripheral.

## 5.5 IMPLEMENTATION

**Communication between TA and PTA.** A user-space TA uses the GlobalPlatform TEE API function `TEE_OpenTASession` to initialise a session with the $I^2S$ PTA using the latter's UUID. After the session is created, the TA uses the `TEE_InvokeTACommand` with a command identifier and related parameters stored in the structure `TEE_Param` to invoke a service, *e.g.*, initialising an $I^2S$ recording operation in the PTA. The PTA in turn invokes the corresponding secure $I^2S$ driver routine, which performs secure I/O operations via MMIO or DMA. The resulting data is transferred securely to the TA in a `TEE_Param memref` buffer. The data is then encrypted (or filtered) inside the TA prior to transmission to the cloud via the TEE supplicant executing inside the REE.

**Key derivation and storage.** OP-TEE features a built-in PTA specifically designed for the derivation of encryption keys based on hardware-unique identifiers, exposed by SoCs that offer such capabilities. These hardware-specific keys are accessible exclusively in the TEE, or are derived differently by the SoC depending on whether they are accessed from the REE or TEE. Within the TEE, the resulting encryption keys are commonly used for encryption and decryption of files, ensuring data security even after device reboots. This functionality in OP-TEE is known as *secure storage* [193]. Other research prototypes serve such uniquely-derived cryptographic materials as the basis for introducing attestation capabilities into TrustZone, fulfilling a gap in Arm's TEE architecture. This feature is essential for establishing the trustworthiness of executing software, thus offering a fundamental security assurance for third-party actors, including cloud service providers [115, 118].

## 5.6 EVALUATION

This section presents an experimental evaluation of FORTRESS. We seek to answer the following questions:

**Q1**: What is the overhead of reading and writing MMIO registers in a TEE? (§5.6.1)
**Q2**: What is the cost of data transfers between the secure driver and a TA? (§5.6.2)
**Q3**: What is the cost of data obfuscation inside a TA? (§5.6.3)
**Q4**: What is the degree of TCB reduction after driver partitioning? (§5.6.4)

**Experimental setup.** Our evaluations were conducted on an Nvidia Jetson AGX Xavier development kit. This kit ships with 8 TrustZone-enabled ARMv8.2 (64 bit) processors (4 active) each clocked at 2265 MHz, and 32 GB of memory. The kit runs Jetson Linux 35.3.1, which is based on Ubuntu 20.04.5 LTS and Linux Kernel 5.10. The OP-TEE OS version used is based on tag `optee_os-nvidia-r35.2.1` of NVIDIA OP-TEE OS fork [126]. Latencies (in clock cycles) are measured by reading the `CNTVCT_EL0` timer register [5]. The frequency of this timer is 31.25 MHz on the experimental system. This timer cannot be accessed directly by a TA; we access its value from TAs via an OP-TEE PTA interface. Unless stated otherwise, we report the average of 100 runs.

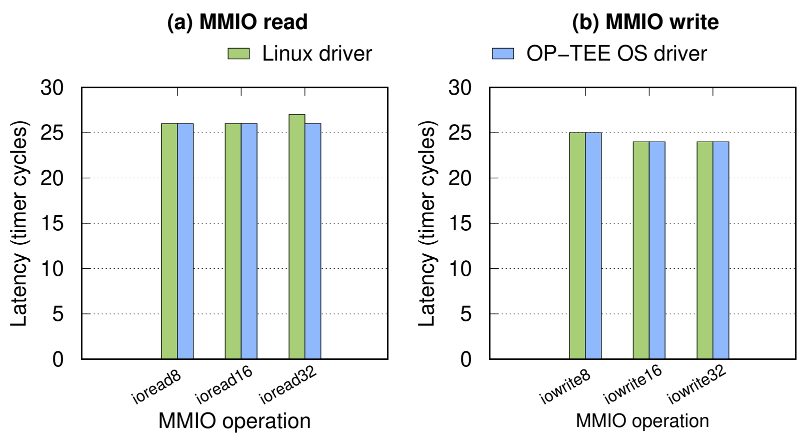


**Figure 5.7:** Cost of MMIO operations.

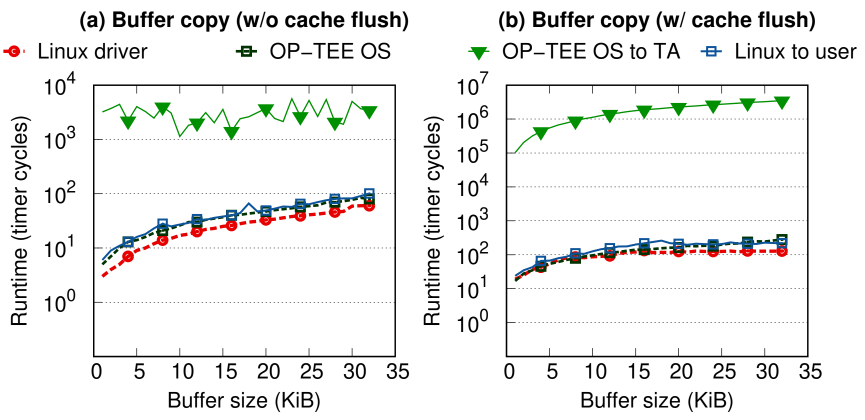


**Figure 5.8:** Cost of buffer copying operations.

### 5.6.1 *Performance of MMIO operations*

*Q1: what is the overhead of reading and writing MMIO registers in a TEE?*

In our initial experiment, we assess the performance of MMIO operations in both secure and normal worlds, using different MMIO APIs, *i.e.*, `ioread{8,16,32}`, `iowrite{8,16,32}`. Our findings, as depicted in Figure 5.7, indicate that MMIO read operations in a secure OP-TEE driver exhibit identical performance as those in a standard normal-world Linux kernel driver. The same observation is true for MMIO write operations. This similarity in performance can be attributed to the fact that MMIO operations share the same underlying implementations in both secure and normal worlds, and are subject to uniform access control policies (*i.e.*, TZASC checks) across both worlds. The read operations have similar performance ($\approx$ 26 cycles) for 8,16, and 32 bits. We also have a similar cost with the write operations ($\approx$ 24 cycles) for 8,16, and 32 bits. This is because the MMIO register width is the same (32-bit) on the device, meaning the full 32 bits are always read or written, and truncated accordingly to align with the exact read or write size.

**Take-away 1**

Kernel driver MMIO operations can be safeguarded within OP-TEE without sacrificing performance.

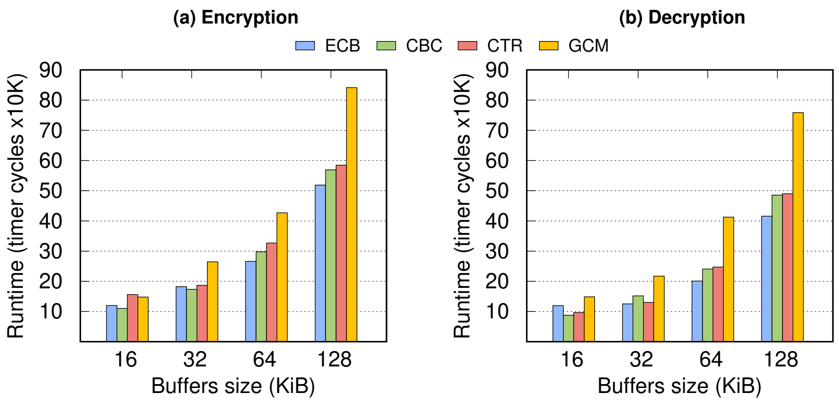


**Figure 5.9:** Cost of cryptographic operations inside a TA.

### *5.6.2 Cost of buffer copying operations*

*Q2: what is the cost of data transfers between the secure driver and a TA?*

This experiment aims to evaluate the cost of transferring data from secure kernel space (OP-TEE OS) to secure user space (TA). It involves the TA invoking an OP-TEE OS driver function through the PTA interface using the function: `TEE_InvokeTACommand`. The TA passes a parameter buffer of type: `TEE_Param` as input, and data is copied (using `memcpy`) from a buffer of the secure driver into the destination buffer. We compare this operation against buffer copy operations from normal world kernel space (Linux driver) to normal world user space, using the `copy_to_user` primitive provided by the Linux kernel. We also evaluate the cost of buffer copy operations in normal and secure world kernel space only, using standard `memcpy` operations. The copy operations are done with and without cache flushing enabled for the source and destination buffers. Figure 5.8 outlines the results, which indicate that copying a buffer from the secure driver in OP-TEE OS to the TA is about 113× and $10K\times$ more expensive on average when compared to copying a buffer from a normal world Linux kernel buffer to a normal world user space buffer, without and with cache flushing respectively. The high overhead observed in the copy operation from OP-TEE OS to the TA is attributed to the additional memory access checks and buffer copying operations involved when transferring data between the TA and OP-TEE OS, as outlined in §5.4.3.

**Take-away 2**

Copying a buffer from OP-TEE OS to a TA is expensive due to additional copy operations and security checks performed during the S-EL0 to S-EL1 context switch.

### *5.6.3 Cost of data encryption/decryption operations*

*Q3: what is the cost of data obfuscation inside a TA?*

In this experiment we evaluate the encryption and decryption costs inside a TA of four AES cryptographic modes of operation: ECB, CBC, CTR, and GCM. All the cryptographic operations use the same 128-bit key. AES-GCM uses an initialisation vector and tag length of 128 bits. Figure 5.9 shows the results obtained.

ECB is relatively simple and deterministic, *i.e.*, identical plaintext blocks produce identical ciphertext blocks; while this mode is relatively cheaper in cost, it is not recommended for securing sensitive data. More advanced algorithms like CBC introduce block chaining, which improves security. CTR mode

**Table 5.1:** Source code % of $I^2S$ driver components in the trusted and untrusted drivers after partitioning.

| Component | % trusted driver | % untrusted driver |
|---|---|---|
| Initialisation | 22.13 | 23.45 |
| Data processing | 41.15 | 0 |
| IRQ handling | 1.33 | 0 |
| Power management | 0 | 7.52 |
| Cleanup | 2.21 | 2.21 |
| **Total** | **66.82** | **33.18** |

has similar performance and security as compared to CBC. GCM mode is most expensive in general because it computes an authentication tag to ensure data integrity, resulting in larger computational overhead. On average, GCM is about 1.3× and 1.6× slower compared to CTR mode for encryption and decryption, respectively. However, in spite of the relatively higher cost, GCM provides both confidentiality and integrity guarantees, while the other modes only ensure confidentility.

**Take-away 3**

Encryption techniques like AES-GCM can be leveraged in the TA to provide confidentiality and integrity guarantees to peripheral data at a reasonable cost.

### *5.6.4 TCB analysis*

*Q4: what is the degree of TCB reduction after driver partitioning?*

Table 5.1 provides a summary of the code components comprising the partitioned $I^2S$ driver, along with the proportion of source code they represent (wrt. the unpartitioned version) in terms of lines of code (LoC). Following the partitioning approach outlined in §5.4.5, 66.82% of the source code was secured inside OP-TEE OS's trusted driver, which corresponds to a 33.18% decrease in the TCB with respect to a system which includes all the source code inside the secure OP-TEE OS driver.

## 5.7 PRACTICAL APPLICATIONS OF FORTRESS

The security architecture provided by FORTRESS can be leveraged in various IoT contexts:

**Smart homes.** In a smart home setup, the central hub can leverage FORTRESS framework to enhance data privacy. FORTRESS acts as a safeguard, stopping the unintentional sharing of sensitive information like surveillance videos or voice recordings to cloud services with data obfuscation. For example, as explained in §5.4.4, FORTRESS can apply data conversion to human speech, converting it into voice commands. This avoids the need to send raw audio (containing potentially sensitive information) for analysis; only the specific voice command can be identified and sent to the cloud service. Implementing such strategies requires moving machine learning-based inference into the TEE, a paradigm that can be achieved with Arm TrustZone [120]. Additionally, peripherals like fingerprint sensors employed for locking mechanisms can be effectively isolated from potential malicious attacks that compromise the OS. The sensor reading, as well as all necessary logic to validate the latter is protected within the TEE.

**Medical IoT.** Similarly to a smart home, medical IoT hubs responsible for aggregating potentially sensitive data from devices such as cardiac monitors can leverage FORTRESS to guarantee that this data is not accessible to a malicious adversary (*e.g.*, on the LAN) who has compromised the OS. Further, the data obfuscation process safeguards the sensitive data once it is transferred to an untrusted cloud service.

## 5.8 RELATED WORK

**Table 5.2:** Comparing FORTRESS with previous approaches.

| | [25] | [23] | [96] | [100] | [209] | **FORTRESS** |
|---|---|---|---|---|---|---|
| Untrusted OS | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| Untrusted hypervisor | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Data confidentiality | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ |
| Data integrity | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| ARM-based | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ |

Several research works have proposed techniques to enhance security guarantees of peripheral data both on ARM and x86-based systems. For ARM-based systems: [106] provides software abstractions to protect the integrity and authenticity of sensor readings; SeCloak [96] leverages TrustZone to provide a way to reliably turn on/off peripherals even in the presence of compromised privileged platform software like the OS or hypervisor, without explicitly providing data confidentiality or integrity guarantees; other systems like [25] leverage an MPU to isolate peripheral memory in MCUs. For x86-based systems: SGXIO [209] and Driverguard [23] leverage a trusted hypervisor to secure sensitive I/O data from an untrusted OS. While these systems propose varied solutions to address I/O data security, they either weaken the threat model by leveraging a trusted OS or hypervisor, or do not provide full confidentiality and integrity guarantees. As summarized in Table 5.2, FORTRESS addresses all these limitations and offers a completely secure path for IoT peripheral data from the hardware to the cloud, safeguarding data confidentiality and integrity.

## 5.9 SUMMARY

This chapter presented FORTRESS, a generic design for safeguarding sensitive peripheral data in IoT environments by leveraging TEE technology. FORTRESS proposes a generic peripheral driver partitioning strategy which restricts peripheral I/O memory regions to a secure kernel-space TEE, thereby limiting access exclusively to a small segment of the driver code. This strategy effectively blocks unauthorised access from potentially compromised operating systems or hypervisors. The data is then securely transferred to a user-space TEE, where obfuscation techniques are applied prior to its transmission to an untrusted host operating system or cloud service provider. Through a proof-of-concept implementation focusing on $I^2S$, our evaluations show that FORTRESS enhances the security posture of IoT devices at a reasonable cost. The security architecture provided by FORTRESS can be used for various applications including smart homes, medical IoT, amongst others.

# Part II

TEE PERFORMANCE ENHANCEMENT

CHAPTER 6

# Improving Enclave I/O Performance with Persistent Memory

The previous chapters focused on optimising TEE security through program partitioning. This chapter shifts the focus to improving performance in TEEs. In that light, we propose techniques to enhance I/O performance of Intel SGX enclaves by leveraging *persistent memory*. We demonstrate the practicality of these techniques by designing and building PLINIUS, a framework for secure and persistent machine learning (ML) model training.

This chapter is organised as follows:

## 6.1 INTRODUCTION

Popular TEE technologies like Intel SGX offer robust security guarantees by providing *enclaves* to shield sensitive code and data deployed in untrusted cloud computing infrastructures. However, enclave security usually comes at a performance cost. A key source of this overhead is I/O operations, for example when reading data from secondary storage into enclave memory, or when writing data to storage to achieve fault tolerance guarantees. Such I/O operations necessitate enclave context switches, *i.e.*, `ecalls` or `ocalls`, which incur substantial overhead [210, 157], leading to performance bottlenecks. This not only affects the efficiency of applications running within SGX enclaves, but also limits the scalability and practicality of enclave applications in high-performance computing scenarios.

*Persistent memory* (PM) offers a promising solution to this problem. Unlike conventional storage technologies, PM combines the speed of DRAM with the non-volatility of storage, providing quicker data access and retention capabilities. Essentially, applications can write (read) data to (from) PM without relying on expensive system calls, *e.g.*, `read`, `write`, *etc*. As such, integrating PM within enclaves can significantly reduce the frequency of (and hence the overhead incurred from) expensive enclave context switches, particularly for applications that depend on (relatively slow) secondary storage, *e.g.*, SSDs, for fault-tolerance guarantees.

While previous research [74] has shown that PM improves performance for a vast number of applications, no research work (at the time of this writing) has proposed PM to improve enclave performance. This work aims to bridge this gap by designing and building tools which can be used to expose PM capabilities to enclave runtimes. On the one hand, we adapt an existing PM library to facilitate the use of PM inside SGX enclaves. On the other hand, we propose a novel approach to ensure efficient fault tolerance guarantees in enclave applications. This approach comprises a *mirroring mechanism* that entails creating encrypted mirror copies of enclave data structures in PM, and synchronising the enclave and PM copies throughout the application's lifetime.

In summary, we make the following contributions:

- We implement and release as open-source SGX-ROMULUS,[1] a PM library for Intel SGX enclaves. SGX-ROMULUS manipulates PM directly from within SGX enclaves, without costly enclave transitions between the trusted and untrusted parts of an SGX application.
- We propose a mechanism to provide efficient fault tolerance guarantees for enclave data structures.
- We leverage SGX-ROMULUS as well as our PM-based fault tolerance mechanism to build PLINIUS, a framework for secure and persistent machine learning model training inside SGX enclaves. PLINIUS is equally released as open-source.[2]
- We provide a comprehensive evaluation of PLINIUS, using real PM hardware and real AWS Spot traces, showing its better performance when compared with traditional checkpointing on secondary storage, *i.e.*, disk or SSD.

## 6.2 BACKGROUND

### *6.2.1 Exposing PM to applications*

There are two ways (summarized in Figure 6.1) to expose PM hardware to applications: (1) as *fast storage* or (2) as *non-volatile byte-addressable memory*.

1 https://github.com/Yuhala/sgx-romulus.git
2 https://github.com/Yuhala/plinius

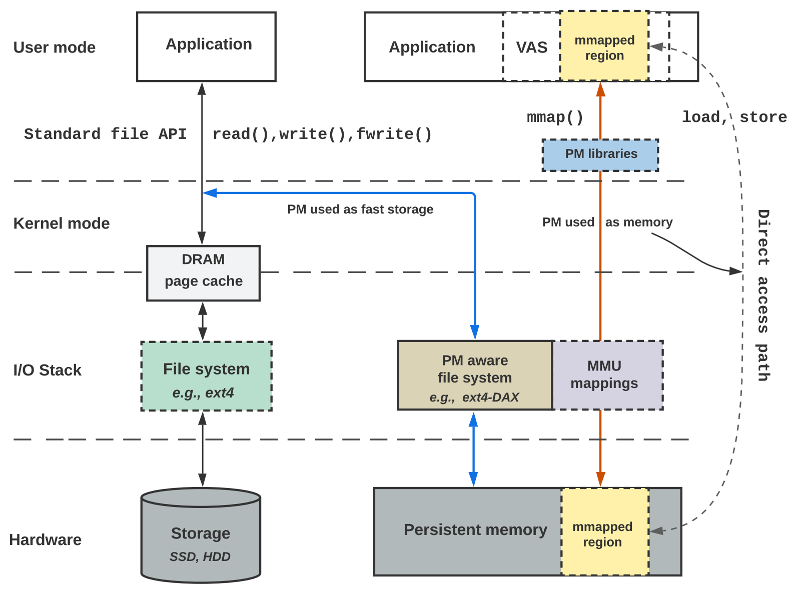


**Figure 6.1:** Exposing persistent memory to applications.

**PM as fast storage.** PM hardware can be exposed to applications as a storage device, and the standard file system API, *e.g.*, `read`, `write`, *etc.* used to manipulate files. This strategy does not require any application changes, and leads to improved performance relative to regular SSDs in many cases [74]. The reason for the increase in performance is because the *page cache* is bypassed for reads and writes to files [74]. The page cache is a portion of DRAM which is generally used to buffer I/O operations in traditional block storage devices like SSDs. For devices such as PM which are byte-addressable like DRAM, the page cache becomes redundant. Nonetheless, leveraging PM as fast storage requires the underlying file system to be made *persistent memory aware*, *i.e.*, by introducing direct access (DAX), an OS feature that removes the page cache from the I/O path, allowing direct access to PM. Some PM aware file systems are Ext4, XFS, NTFS, and NOVA [215]. While exposing PM as fast storage leads to some performance improvements, this strategy does not lead to optimal performance because the file system adds some overhead to the I/O path.

**PM as non-volatile byte-addressable memory.** To achieve optimal performance with PM, an application must be modified to access the former as byte-addressable memory. This is most commonly done by *memory mapping* PM into the application's VAS, and using `load`/`store` instructions to access PM without any intervention of the underlying file system. This strategy offers the shortest path from the application to PM. The memory mapping is done using standard calls like `mmap()` on Linux or `MapViewOffile()` on Windows [166]. This creates a byte-for-byte correlation between the mapped portion of the VAS and the file in PM, thereby enabling direct `load`/`store` from/to the memory mapped file, *e.g.*, via regular pointers and routines such as `memcpy`, `strcpy`, *etc.* As shown in Figure 6.1 the DAX feature in the persistent memory aware file system allows mapping of PM page frames into the process's page table which is used by the MMU during address translation.

It is worth noting that memory mapping can also be performed on files in regular storage devices like SSDs. However, the key difference is the memory mapped file on an SSD exists in the page cache and all changes need to be synchronized with the actual file on SSD; this is done via an `msync` call which writes the modified pages (*i.e.*, 4*KB* granularity) to the underlying storage device. With PM on the other hand, the page cache is removed from the I/O path, and the data is read/written directly from/to PM at byte granularity.

**Table 6.1:** Comparing the sizes of PM libraries.

| PM library | LoC |
|---|---|
| Romulus | 18,100 |
| PMDK | 131,000 |

#### 6.2.1.1 *Durability and consistency*

At the *x86* instruction level, simply executing a `store` instruction does not guarantee that the data has reached PM [166]. To ensure durability of data in PM, the application must issue a *flush* instruction to move the data from the cache line to PM. Additionally, a *fence* operation is used to enforce ordering in flush instructions; this ensures data consistency. The *x86* instruction set provides two flush instructions: `CLFLUSHOPT` (cache line flush optimized) and `CLWB` (cache line write back). `CLW` differs from `CLFLUSHOPT` in that `CLWB` keeps the data in the cache whereas `CLFLUSHOPT` does not [166]. Similarly, x86 provides persistence fences which guarantee consistency by preventing `store` instructions from being re-ordered by the CPU. For example, `MFENCE`, which orders `load`, `store`, and `flush` instructions; and SFENCE, which orders `store` and `flush` instructions [125]. The *x86* instruction set provides a third instruction, `CLFLUSH`, which operates as both a flush and fence for the cache line being flushed, *i.e.*, flushes are ordered without the need for explicit fence instructions like `SFENCE` or `MFENCE`.

### ***6.2.2 Persistent memory libraries***

As one might expect, the improved performance of PM relative to conventional storage comes at the cost of increased complexity for the application developer, because it requires a paradigm shift in the way applications are built, *i.e.*, explicit flush/fence instructions, ensuring consistency across system failures, *etc*. As a result, several software tools and libraries have been proposed to facilitate PM development. Examples include Intel's persistent memory development kit (PMDK) [158], Romulus [33], Mnemosyne [202], Pangolin [227], amongst many others. PM libraries expose PM to applications by memory-mapping files on a persistent memory-aware file system with DAX capabilities, and provide an API which enables developers to easily create and manipulate data structures in PM.

These libraries use the aforementioned processor instructions, *i.e.*, `CLFLUSH`, `CLFLUSHOPT`, `CLWB`, to flush data from cache lines to PM. Additionally, these libraries provide logging techniques, *e.g.*, *undo* or *redo* logs to ensure atomic updates to PM. As such, in the event of a system crash or power failure, the PM application can rollback to a previous consistent state (undo logging), or replay the failed update (redo logging).

### ***6.2.3 Using PM libraries inside SGX enclaves***

To utilize PM efficiently within enclave environments, it is essential to incorporate a PM library into the enclave runtime. Since the aforementioned PM libraries were not designed with enclaves in mind, they must either be run inside enclaves with the help of a LibOS like Gramine [195] or a tool like SCONE [7], or refactored for an enclave environment. As discussed in previous chapters, an important factor to be taken into consideration with enclave software is the TCB. Using the LibOS approach entails including the entire PM library, the LibOS itself, and potentially the full GNU standard C library (libc) inside the enclave, significantly increasing the TCB. Alternatively, by refactoring the library, *e.g.*, through code partitioning, one could decrease the amount of code that is ultimately included in the enclave. We take the latter approach, and adapt an existing PM library for SGX enclaves. Given the variety of PM libraries, the size of these libraries should also be taken into

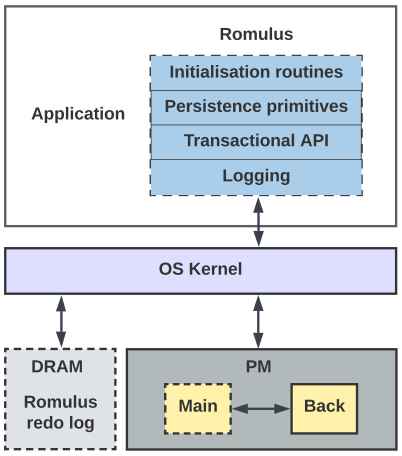


**Figure 6.2:** Architecture of Romulus.

account. In Table 6.1, we compare the size (in LoC) of two open-source PM libraries: the PMDK and Romulus. While the PMDK is more popular, it is prohibitively too large and complex for an enclave environment. In constrast, Romulus is about 7× smaller than the PMDK, and provides an adequate API for PM development. Consequently, it is a better choice for an enclave environment.

**Architecture of Romulus.** The overall design of Romulus (outlined Figure 6.2) is straightforward, and can be subdivided into four main components: PM initialisation routines, persistence primitives, transactional API, and logging. The initialisation routines permit to expose PM hardware to the application via the aforementioned memory mapping technique. Persistence primitives comprise the API for writing data to PM via `CLFLUSH`, `CLFLUSHOPT` or `CLWB` operations, as well as memory barrier operations for consistency. To ensure atomic updates to PM, Romulus provides durable transactions via twin copies of data in PM, and relies on a volatile log to track memory locations being modified in a transaction. The first copy, called the `main` region, is where user-code executes all in-place modifications; the second copy, the `back` region, is a *backup* (or snapshot) of the previous consistent state of the `main` region. Following a crash while carrying out modifications ("mutating") in the `main` region, the content of `back` is restored to `main`. If no crash occurs during these modifications, the content of `main` becomes the most consistent state, and is copied to `back` when the transaction completes. In the following section, we adapt Romulus for Intel SGX enclaves, hence SGX-ROMULUS.

## 6.3 SGX-ROMULUS ARCHITECTURE

SGX-ROMULUS is Romulus tailored for Intel SGX enclaves. As previously discussed, adapting software for an enclave runtime typically involves partitioning. In the case of Romulus, this means identifying which parts of the library handle sensitive information; these components are isolated within the enclave. It is logical to expect that enclave software leveraging PM as byte-addressable memory deals with confidential data which should not be exposed. To meet this security requirement, all Romulus components for reading, writing, or logging data should be confined within the enclave. Unlike regular enclave pages, PM pages accessed by the enclave do not undergo automatic encryption/decryption by the MEE. As a result, application-level encryption techniques must be employed. On the other hand, the initial memory mapping of PM into the application's address space cannot be done inside the enclave because it relies on unsupported system calls like `mmap`, `read`, `write` *etc*. Nonetheless, doing the memory mappings out of the enclave runtime does not compromise the confidentiality of data in PM if it is encrypted.

The architecure of SGX-ROMULUS is summarized in Figure 6.3. The in-enclave user-space library,

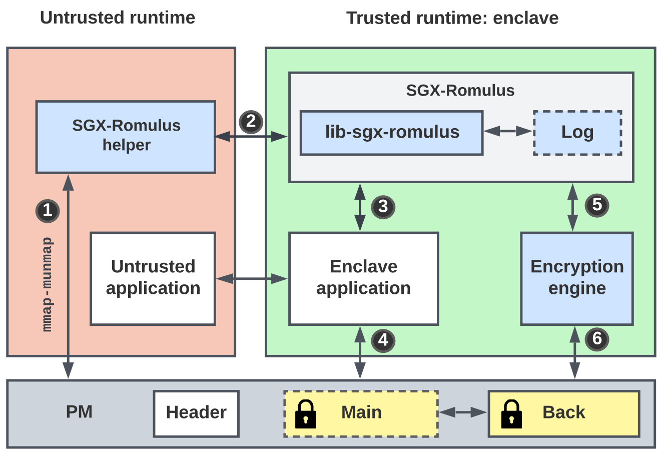


**Figure 6.3:** Architecture of SGX-ROMULUS.

`lib-sgx-romulus`, provides persistence primitives and a transactional API required to create and manage persistent data structures in PM. The encryption engine permits to encrypt data written to PM and decrypt data read from PM. It is based on AES Galois Counter Mode (GCM) [39], a cryptographic algorithm that provides both confidentiality and integrity guarantees. Similar to Romulus, SGX-ROMULUS maintains a volatile log in enclave memory (rather than DRAM) to record the addresses and ranges of modified data in the current transaction. A helper library in the untrusted runtime, `sgx-romulus-helper`, interfaces with SGX-ROMULUS and performs necessary system calls (*e.g.*, `mmap`, `unmap`) which are required when leveraging PM via a DAX-enabled file system.

At application initialisation , `sgx-romulus-helper`[3] memory-maps the file corresponding to the persistent memory region (*main*, *back*, and *header*) into the application's VAS, via a `mmap` system call (❶). Then, `sgx-romulus-helper` initialises the *persistence header* [33], which holds metadata to track the consistency state of the *main* and *back* regions, a reference to an array of persistent memory objects, and a pointer to the memory allocator's metadata, *e.g.*, allocated and unallocated PM. The address of the persistence header is passed to `lib-sgx-romulus`[4] via an `ecall` (❷), and once the enclave validates this address,[5] it then completes the PM region initialisation, as detailed in Algorithm 3. The enclave can then create or update persistent data structures in PM (❸-❹). If confidentiality is required for PM data, the cryptographic API provided by the encryption engine is leveraged (❺) and the encrypted data flushed to PM (❻). The keys used for encryption can be provided to the enclave via SGX's remote attestation mechanism. Upon graceful termination of the enclave application, the enclave runtime issues an `ocall` to unmap the PM region from application VAS via the `munmap` system call.

### 6.3.1 *Fault tolerance in enclaves*

A useful application for PM inside enclaves is to achieve fault tolerance for I/O heavy applications, for example ML model training that employs frequent checkpointing. The API provided by SGX-ROMULUS can be levereged to provide robust fault tolerance capabilities for enclave applications. We propose an approach called *mirroring* which involves creating encrypted copies of enclave data structures in PM and synchronising the enclave and PM copies throughout the application's lifetime. In the event of a crash or system failure, the copy in PM is retrieved and decrypted inside the enclave, allowing the application to proceed from a consistent state prior to the crash or system failure.

3 SGX-ROMULUS helper
4 Passing persistence header to the enclave
5 Validating persistence header

**Algorithm 3** SGX-ROMULUS initialisation.

```
## Untrusted (outside of enclave) ##
function init_sgx_romulus(pm_file)
  mapped_addr = mmap(pm_file)
  header_addr = create_header(mapped_addr)
  ecall_init(header_addr)
end
function ocall_unmap
  munmap(pm_file)
end
## Trusted (inside enclave) ##
function ecall_init(header_addr)
  initialize_main_and_back(header_addr)
  recover() // recovery operations in case a crash occurred [33]
end function
```

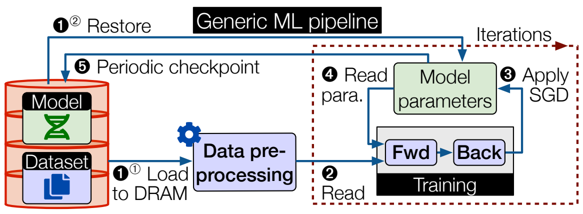


**Figure 6.4:** Model training: *Fwd*=Forward propagation, *Back*=Backward propagation.

In the following section, we explore how this concept can be applied to privacy-preserving machine learning, and demonstrate its practicality by designing and implementing a robust PM-enabled ML framework for enclaves.

## 6.4 MACHINE LEARNING IN A NUTSHELL

Machine learning involves developing algorithms that can analyse large sample datasets and build mathematical models that capture the underlying patterns and relationships within these datasets. A ML *model* can be described as a function that maps an input to a target output based on a set of parameters [68]. For instance, a linear regression model can be represented as: $\mathbf{f}(\mathbf{x}) = \mathbf{W}^\mathsf{T}\mathbf{x} + \mathbf{b}$, where $\mathbf{W}$ represents the model weights, $\mathbf{b}$ the bias vector, and $\mathbf{x}$ the input vector. The weights and biases are the learnable parameters of a model, and are determined through a process called *model training*.

The goal of model training is to obtain the set of learnable parameters that minimises a *loss function* (*e.g.*, via stochastic gradient descent (SGD) [123]) and maximizes the model's accuracy on some test data. The loss function is a scalar function that quantifies the difference between the predicted value (for a given input data point) and the ground truth or real value [1]. As illustrated in Figure 6.4, during training, the learning algorithm iteratively feeds the model with batches of training data read from storage into DRAM (❶$^{①}$), calculates the loss, and updates the model parameters in such a way as to minimise the loss (❷-❸-❹). Training models such as deep neural networks can take up to several days, a time window sufficiently long for training jobs to experience failures or pre-emptions [1]. In the event of a failure during training, the model being trained as well as the training datasets resident in DRAM are lost and need to be retrieved from secondary storage upon restart (❶$^{②}$). As a result, several state-of-the-art ML frameworks, *e.g.*, Tensorflow [1], Darknet [36], Caffe [78], *etc.* provide mechanisms to checkpoint model states to secondary storage during training (❺). This enables the training process to resume from the most recent checkpoint, rather than from scratch. Technologies like PM can be leveraged to reduce the cost of the checkpoint and restore process. For example, training data or a model resident in PM will persists across failures.

**Privacy-preserving machine learning.** It is an emerging ML paradigm focused on developing ML models while safeguarding confidential training data. This approach is particularly vital today as using (untrusted) third-party cloud services for large-scale ML jobs has become the de facto standard [176]. Techniques like differential privacy [184, 76], federated learning [82], and homomorphic encryption [201] have been employed to mitigate this problem.

Recently, the concept of confidential models [190] has been gaining traction. This approach goes beyond just protecting the training data; it also focuses on safeguarding the details of the ML models themselves, *e.g.*, weights and biases. Confidential models are essential in situations where the model contains proprietary information or represents intellectual property that could offer competitive advantages, or simply contain sensitive information. While the aforementioned techniques like homomorphic encryption can be employed, the substantial overhead they introduce has led to a shift towards TEE technologies like Intel SGX, Intel TDX, AMD SEV *etc.* to achieve efficient confidentiality guarantees.

Nonetheless, TEE-based techniques introduce some bottlenecks, particularly regarding I/O. On the one hand, large quantities of training data need to be read from secondary storage to enclave memory (volatile), necessitating frequent and expensive enclave transitions. On the other hand, ML jobs tend to experience frequent interruptions, *e.g.*, when deployed on virtual machine instances such as EC2 spot instances [205]. Such jobs rely on regular checkpointing to secondary storage, which equally necessitates expensive transitions when performed in an enclave runtime.

We posit that a tool like SGX-ROMULUS, along with our mirroring mechanism, offer a promising solution to address these challenges. In this context, we have designed and built PLINIUS, a framework to achieve secure and persistent ML model training inside enclaves. PLINIUS leverages the mirroring mechanism to maintain a persistent encrypted copy of a ML model being trained in PM; in case of a system crash or failure, this copy is retrieved into the enclave (without the need for expensive enclave transitions for I/O), decrypted, and used to reconstruct the enclave model. Additionally, PLINIUS maintains confidential training data in PM, eliminating the need to access secondary storage in the event of a system failure or crash.

## 6.5 PLINIUS THREAT MODEL

PLINIUS has three security goals:

- **G1:** Ensure confidentiality and integrity of training data in byte-addressable PM.
- **G2:** Ensure confidentiality and integrity of a ML model's parameters (*e.g.*, weights, biases) during training.
- **G3:** Ensure confidentiality and integrity of the model's replica in PM.

The system is designed to achieve these security goals while facing a powerful adversary with physical access to the hardware and full control of the entire software stack including the OS and hypervisor. The adversary seeks sensitive information inside the enclave, on DRAM or PM, or data from the processor.

Model hyper-parameters such as model architecture, number of layers, size of training batches or type of training data are usually public information, as they do not leak any information about trained model parameters or sensitive training data [59, 68, 65]. In order to mitigate possible threats linked to malicious data sources, PLINIUS supports secure provisioning of model hyper-parameters via the SGX remote attestation mechanism.

We assume that the adversary cannot physically open and manipulate the processor package, and that

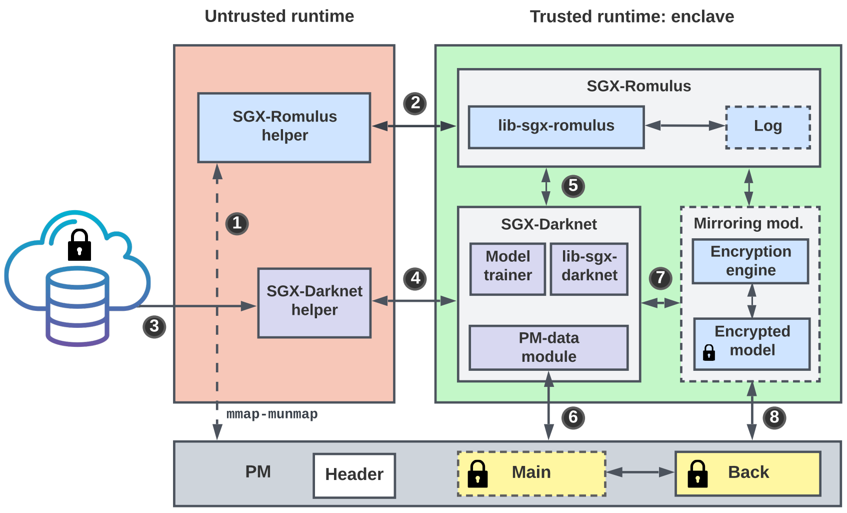


**Figure 6.5:** PLINIUS architecture.

enclave code is correct and does not leak sensitive information (*e.g.*, encryption keys) intentionally. Denial-of-service and side-channel attacks [16, 171], for which solutions exist [128, 60], are considered out of scope.

## 6.6 PLINIUS ARCHITECTURE

PLINIUS is an enclave-based ML machine learning framework which leverages PM for fault tolerance capabilities. It is based on Darknet [36], a popular open-source ML library. The architecture of PLINIUS consists of three main components interacting with each other: *(1)* an SGX-compatible deep-learning framework, *i.e.*, SGX-DARKNET; *(2)* an SGX-compatible PM library, *i.e.*, SGX-ROMULUS; *(3)* a *mirroring module*, which synchronizes the ML model inside the enclave with its encrypted mirror copy in PM. Figure 6.5 shows how these components interact.

**Design overview.** At system initialisation, SGX-ROMULUS helper memory maps PM into the application's VAS (❶), and initialises SGX-ROMULUS (❷). Training data initially in secondary storage (*i.e.*, SSD) is first read into DRAM (❸), and loaded into PM (❹-❺-❻) via a PM data module. The data is then used to train a model inside the enclave with the help of SGX-DARKNET, while the mirroring mechanism periodically encrypts and saves the trained model to PM with the help of SGX-ROMULUS and the mirroring module (❼-❽). Following a system crash or power failure, training is resumed and encrypted model in PM is retrieved and decrypted in enclave memory (❽).

**SGX-Darknet.** It is an SGX-compatible version of Darknet [36] ML framework. To maintain a small TCB, we partition SGX-DARKNET into a trusted and untrusted component. Our partitioning strategy involves keeping out of the enclave all computations which are not critical for upholding security guarantees outlined in our threat model. For example, tasks such as parsing model configuation files to determine the structure of a ML model are kept outside the enclave. We also strip all GPU code (in CUDA) from SGX-Darknet as GPUs do not support Intel SGX at the time of this writing. To minimise code alterations for commonly used (but unsupported) routines in Darknet (*e.g.*, `fread`, `fwrite`, *etc.*), SGX-DARKNET redefines these routines as wrapper functions for `ocalls` to the corresponding libc functions in the untrusted runtime. A support library in the untrusted runtime, *SGX-darknet-helper*, provides the implementations of those `ocalls`, invoking the corresponding libc routines. The *pm-data-module* leverages SGX-ROMULUS to read (write) encrypted datasets from (to) PM. Finally, *lib-sgx-darknet* provides the API to train and do inference on models from within the enclave runtime.

**Mirroring module.** This component is in charge of creating and updating encrypted mirror copies of enclave ML models in PM. It contains the necessary logic to instantiate models that are both persistent and directly byte-accessible via loads and stores. It leverages the transactional API provided by SGX-ROMULUS to perform atomic updates on persistent models in PM. This is crucial as it prevents any inconsistency in PM data structures in the event of a system failure during data updates. This module performs two main operations: *mirror-out*, which involves encrypting an enclave data structure and flushing it to PM, and *mirror-in* which involves reading an encrypted data structure from PM (via a simple `memcpy`) and decrypting it in enclave memory. The encryption engine is responsible for all the the encryption and decryption operations. It uses a 128, 192 or 256 bit key for all cryptographic operations, and provides assurance of the integrity of the confidential data. As recommended by [40], for every encryption operation, a random 12-byte *initialisation vector* (IV) is generated for each encryption operation using the `sgx_read_rand()` [31, p. 200] function from the Intel SGX SDK. The encryption algorithm divides each plain text buffer into 128 bit blocks which are encrypted via AES-GCM. The `IV` and a 16-byte message authentication code (MAC) are then appended to each encrypted data buffer. The MAC is used to ensure data integrity during decryption.

The key used for encryption/decryption can be provisioned to the enclave via remote attestation [31, p. 99] or could be generated securely (*e.g.*, if training data is not encrypted) inside the enclave using `sgx_read_rand()`. The encryption key, once generated or provisioned, can be securely sealed [31, p. 96] by the enclave for future use.

### *6.6.1 Model training in* PLINIUS

The workflow for ML model training in PLINIUS can be divided into three main phases: *PM initialisation*, *dataset loading*, and *model training and mirroring*.

**PM initialisation.** This phase is equivalent to the PM initialisation phase for SGX-ROMULUS, as described in Algorithm 3. It memory maps PM into the application's VAS and initialises the persistent regions *main* and *back* so that both regions are consistent before the training algorithm starts.

**Inital dataset loading into PM.** A key feature of PLINIUS is its capability to use training data in PM, bypassing the need to read data from secondary storage into volatile memory (DRAM). In PLINIUS, training data is loaded into PM once, after which the data stays in (byte addressable) PM. Following a system crash or power failure, training data in PM is almost immediately accessible to the training algorithm, unlike in disk or SSD-based systems where data must be reloaded from slower secondary storage back into DRAM.

Initially, the training dataset exists as encrypted files in secondary storage. Darknet training algorithms operate on input data formatted as multidimensional arrays or matrices. To leverage the byte-addressable nature of PM, the training data should be loaded into such a data matrix in PM. *SGX-Darknet-helper* facilitates this process by transferring training dataset and labels from secondary storage into DRAM, forming a volatile matrix variable. The address of this matrix is sent to SGX-DARKNET through an `ecall`. Subsequently, the *PM-data-module* uses `lib-sgx-romulus` to create a corresponding persistent matrix in PM. Once the persistent matrix is created, the encrypted training data is simply `memcpy`-ied from DRAM into PM within a Romulus transaction inside the enclave. The persistent data can then be accessed directly via its address.
During training the training data is `memcpy`-ied from PM into enclave memory where it is then securely decrypted. This initial data loading step completely achieves **G1**.

**Model training and mirroring.** The PLINIUS architecture fits well for training neural network models[123], which consist of multiple *layers* with learnable parameters, *i.e.*, weights and biases. The

**Algorithm 4** Persistent model allocation and training

```
1  function alloc_mirror_model()
2    // Romulus transaction start
3    BEGIN_TRANSACTION [33]
4      // allocate layer
5      head_pm_layer = PMalloc(size)
6      // allocate layer's parameters
7      head_pm_layer.W = PMalloc(size)
8      head_pm_layer.next = nullptr
9      cur_pm_layer = head_pm_layer
10     // num. of layers in enclave model
11     n = enclave_model.numL
12     for i = 2 to n do
13       cur_pm_layer.next = PMalloc(size)
14       cur_pm_layer = cur_pm_layer.next
15       cur_pm_layer.W = PMalloc(size)
16       cur_pm_layer.next = nullptr
17     end for
18   // Romulus transaction end
19   END_TRANSACTION [33]
20 end function
21 function train_model(config)
22   enclave_model = create_enc_model(config)
23   if not_exists(pm_data) then
24     ocall_load_data_in_pm()
25   end if
26   iter = 0
27   if exists(pm_model) then
28     mirror_in(enclave_model)
29     iter = pm_model.iter
30   else
31     pm_model = alloc_mirror_model()
32   end if
33   while iter < MAX_ITER do
34     data_batch = decrypt_pm_data(size)
35     train(enclave_model,data_batch)
36     mirror_out(enclave_model,iter)
37   end while
38   free(enclave_model)
39 end function
40
```

architecture of the model and its hyper-parameters (*e.g.*, layer types, batch size, learning rate, *etc.*) are defined in a configuration file which is parsed into a `config` data structure by *SGX-Darknet-helper* in the untrusted runtime. The address of the `config` data structure is sent to the enclave via an `ecall` where it is used to build the enclave model. PLINIUS represents a neural network model in PM as a linked list of persistent layer structures, which simplifies future modification of the model's structure, *e.g.*, adding or removing layers. The model's layers contain persistent attributes, *e.g.*, weight vector, bias vector, *etc.* By creating the ML model in enclave memory, we achieve **G2**.

Algorithms 4 and 5 summarize respectively model training and mirroring in PLINIUS. If the training dataset has not been loaded in PM, an `ocall` is performed to load data from secondary storage into PM (Algorithm 4 line 24) as described previously. If a persistent mirror model exists in PM, we *mirror-in* (read from PM and decrypt in enclave) its parameters into the enclave model (Algorithm 4 line 28), otherwise we allocate one in PM as described in Algorithm 4.

During model training, batches of training data are decrypted from PM (Algorithm 4, line 34) into enclave memory and used to train the enclave model for one training iteration. After each training iteration PLINIUS does a *mirror-out* operation which encrypts the model parameters and writes them to the model's mirror copy in PM (Algorithm 4 line 36), thus synchronising the enclave and PM models, and hence achieving **G3**.

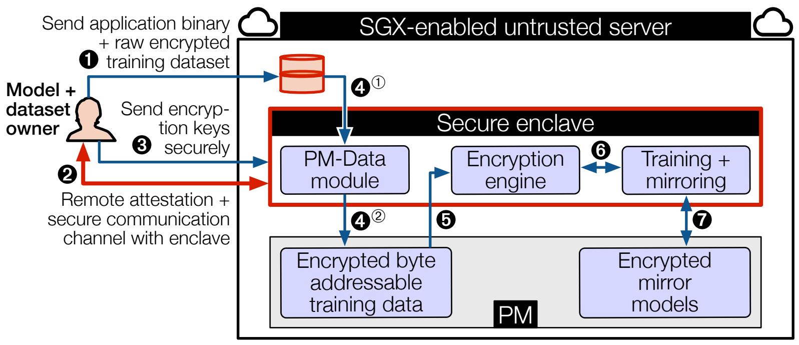
In the event of a crash during training, upon resumption the model and training data are already in PM and can be quickly `memcpy`-ied from PM into secure enclave memory. This obviates the need for much more slower read operations from storage devices like SSDs and HDDs.

**Synthesis.** Figure 6.6 shows the overall ML workflow with PLINIUS. The owner of the data and the model sends the application binary and raw encrypted training data to the remote untrusted server (Figure 6.6-❶). She then performs remote attestation (RA), establishes a secure communication channel (SC) with the enclave (Figure 6.6-❷) and sends encryption keys to the latter (Figure 6.6-❸). The PM-data module transforms encrypted data on disk to encrypted byte addressable data in PM (Figure 6.6-❹$^{①,②}$). The training module reads and decrypts (with keys obtained from RA & SC) batches of training data from PM (Figure 6.6, ❺-❻) with the trained model being mirrored to PM or into the enclave for restores (Figure 6.6-❼).

**Algorithm 5** Mirroring algorithms

```
function mirror_out(enclave_model,iter)
  BEGIN_TRANSACTION
  n = enclave_model.numL
  pm_model.iter = iter
  temp = head_pm_L
  for i = 1 to n do
    temp.W  = encrypt(enclave_model.L(i).W)
    temp = temp.next
  end for
  END_TRANSACTION
end function
```

```
function mirror_in(enclave_model)
  n = pm_model.numL
  temp = head_pm_L
  for i = 1 to n do
    enclave_model.L(i).W = decrypt(temp.W)
    temp = temp.next
  end for
end function
```



Send application binary + raw encrypted training dataset
❶
SGX-enabled untrusted server
Model + dataset owner
Send encryption keys securely
❸
❷
Remote attestation + secure communication channel with enclave
❹①
Secure enclave
PM-Data module
Encryption engine
❻
Training + mirroring
❹②
❺
❼
Encrypted byte addressable training data
PM
Encrypted mirror models

**Figure 6.6:** Full model training workflow with PLINIUS.

**Integration with different ML libraries.** The current PLINIUS architecture uses Darknet as the ML library, due to its efficient and lightweight implementation in C that facilitates integration with SGX enclaves. Other ML libraries could be integrated into the PLINIUS architecture. In fact, once the ML library is ported to SGX, the same PLINIUS architecture holds.

## 6.7 IMPLEMENTATION DETAILS

We implement PLINIUS in C and C++; it comprises 28,450 lines of code in total, the trusted portion being 15,900 LoC. We use Intel SGX SDK v2.8 for Linux. The total size of application binary including the enclave shared library after compilation is 3 MB.

## 6.8 EVALUATION

Our experimental evaluation of PLINIUS aims to answer the following questions:

**Q1**: How does PM affect enclave model save and restore performance when compared to SSD? (§6.8.2)

**Q2**: How scalable is PLINIUS with varying model sizes? (§6.8.2)

**Q3**: How robust is the mirroring mechanism against crashes? (§6.8.3)

### 6.8.1 *Experimental setup.*

**Servers.** At the time of this writing, servers that support both SGX and PM are yet to be released. Hence, we adopt a dual setup comprising two experimental nodes (*i.e.*, servers):

Node 1: *SGX-emlPM*, supports SGX but has no physical PM, hence we resort to emulating the latter with

Ramdisk.[6] This machine is equipped with a quad-core Intel Xeon E3-1270 CPU clocked at 3.80 GHz, and 64 GB of DRAM. The CPU ships with 32 KB L1i and L1d caches, 256 KB L2 cache and 8 MB L3 cache.

Node 2: *emlSGX-PM*, is equipped with 4× Intel OptaneDC PM DIMMs of 128 GB each. However, its processors lack native support for SGX. Hence, we resort to SGX in simulation mode [31]. The *emlSGX-PM* node is a dual-socket 40-core Intel Xeon Gold 5215 clocked at 2.50 GHz and 376 GB of DRAM. Each processor has 32 KB L1i and L1d caches, 1 MB L2 cache and a shared 13.75 MB L3 cache.

All nodes run Ubuntu 18.04.1 LTS 64 bit and Linux kernel 4.15.0-54. We run the Intel SGX platform software, SDK and driver version v2.8. All our enclaves have max heap sizes of 8 GB and stack sizes of 8 MB. The EPC size is 128 MB of which 93.5 MB are usable by enclaves. Unless stated otherwise, we use `CLFLUSHOPT` and `SFENCE` for persistent write backs and ordering.

By using a dual setup, we highlight the performance implications of both real SGX and real PM. All experimental comparisons are executed separately for each server, as they have completely different characteristics. We indicate where necessary on which node an experiment is carried out.

**Machine learning.** All models used in our evaluations are convolutional neural networks (CNNs). The convolutional layers use *leaky rectified linear unit* (LReLU) [59] as activation, and all output layers are *softmax* [123] layers. The model optimisation algorithm used is stochastic gradient descent (SGD), and the learning rate used is 0.1. Except stated otherwise, all training iterations use a batch size of 128. Concerning the dataset, we use MNIST [185], a popular dataset in the deep learning community. It consists of 70,000 grayscale images of handwritten digits: 60,000 training samples and 10,000 test samples.

### 6.8.2 PM mirroring vs. SSD-based checkpointing

*Q1 and Q2: how does PM affect enclave model save and restore performance for varying model sizes?*

Here we compare the mirroring mechanism in PLINIUS to traditional checkpointing on SSD using SGX-DARKNET. For SSD checkpointing, we use `ocalls` to `fread` and `fwrite` libc routines to read/write from/to SSD. After each call to `fwrite`, we flush the libc buffers and issue an `fsync`; this guarantees data is actually written to secondary storage and ensures a fair comparison against PM, where data is equally flushed from cachelines into PM. We vary model sizes by increasing the total number of convolutional layers. We measure the times to save/`mirror-out` (encrypt in the enclave and write to PM) and restore/`mirror-in` (read from PM into enclave and decrypt), and compare these to SSD-based checkpoint saves (encrypt and write to SSD) and SSD-based checkpoint restores (read from SSD into enclave and decrypt), which are the state-of-the-art methods for fault tolerance. All data points are an average of 5 runs.

Figure 6.7 represents the results obtained on our two servers. As a general observation, in PLINIUS, in-enclave data encryption contributes more to the overhead of saves (*i.e.*, mirror-out) when compared to writes to PM. For restores in PLINIUS, reads from PM into enclave memory contribute more to the overall overhead.

Table 6.2a shows a performance breakdown of each mirroring steps for saves and restores in PLINIUS, while Table 6.2b shows the average performance improvements of our mirroring mechanism when compared to SSD-based checkpointing. To reduce the effect of outliers, we evaluate results beneath

6 `https://pmem.io/blog/2016/02/how-to-emulate-persistent-memory/`

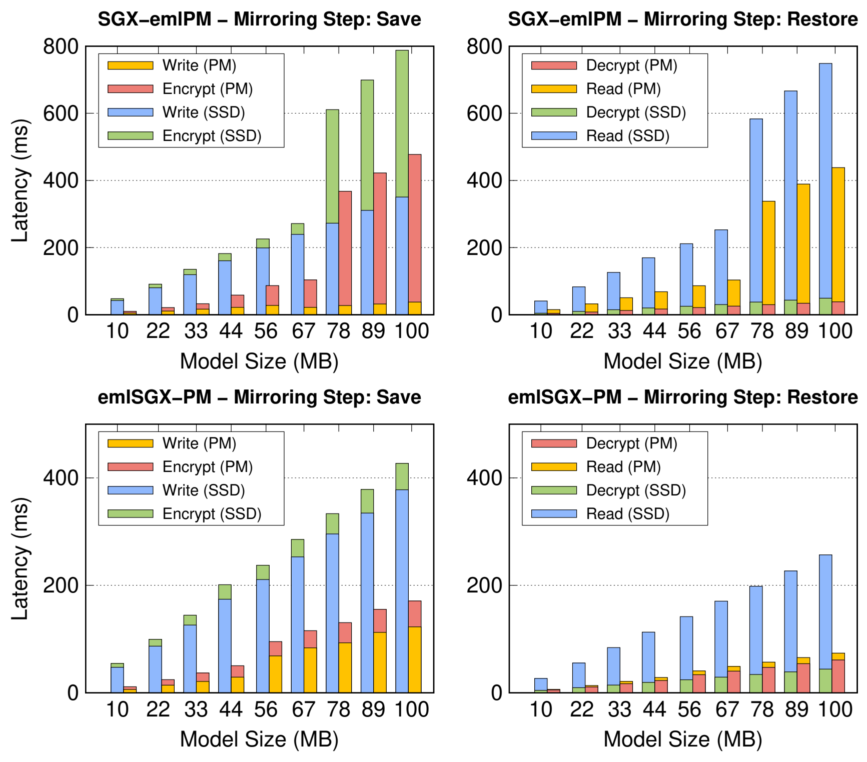


**Figure 6.7:** PM mirroring vs. checkpointing on SSD for SGX-emlPM (top) and emlSGX-PM (bottom).

**(a)** Contributions to the overall runtime (%)

| Save | SGX-emlPM | emlSGX-PM |
|---|---|---|
| Encrypt | 66.4% | 30.3% |
| | 92.3% | |
| Write | 33.6% | 69.7% |
| | 7.7% | |
| **Restore** | **SGX-emlPM** | **emlSGX-PM** |
| Read | 75% | 17.8% |
| | 91.2% | |
| Decrypt | 25% | 82.2% |
| | 8.8% | |

**(b)** Speed-up achieved with PLINIUS wrt. SSD

| Save | SGX-emlPM | emlSGX-PM |
|---|---|---|
| Write | 7.9× | 4.5× |
| | 9.6× | |
| Total | 3.5× | 3.2× |
| | 1.7× | |
| **Restore** | **SGX-emlPM** | **emlSGX-PM** |
| Read | 3× | 16.8× |
| | 1.8× | |
| Total | 2.5× | 3.7× |
| | 1.7× | |

**Table 6.2:** Save and restore performance of PM vs. SSD. Shaded cells indicate values beyond the EPC size.

and beyond the EPC limit separately. The usable EPC size is 93.5 MB, reached for model size 78 MB, due to the presence of other data structures in enclave memory (*e.g.*, temporary buffers used for encryption) as well as enclave code.

We observe (in Table 6.2a) that for saves in a real SGX environment, encryption contributes more (66.4%) to the overall mirroring latency on average for model sizes beneath 78 MB. This increases to 92.3% once the EPC limit is exceeded. This overhead is attributed to expensive page swapping operations between the EPC and regular DRAM by the SGX Linux kernel driver. Regarding restores, reads contribute on average 75% and 91.2% for values beneath and beyond the EPC limit respectively. Similarly, there is substantial overhead beyond the EPC limit due to EPC page swapping. The results suggest in-enclave decryption is relatively cheaper.

For the *emlSGX-PM* server, without real SGX hardware (hence no expensive page swaps), the primary bottleneck is real PM. We observe (in Table 6.2b) that, for the server *SGX-emlPM*, writes to PM are on average 7.9× and 9.6× faster when compared to writes to SSD for enclave sizes beneath and beyond the EPC limit respectively. SSD writes are generally more expensive due to expensive `ocalls` and serialisation operations to secondary storage. Saves are overall 3.5× and 1.7× faster for enclave sizes beneath and beyond the EPC limit respectively. Similarly, for restores, reads from PM into enclave memory are on average 3× and 1.8× faster for enclave sizes beneath and beyond the EPC limit respectively, when compared to the SSD-based counterpart. Restores are overall 2.5× and 1.7× faster for enclave sizes beneath and beyond the EPC limit. A similar breakdown is done for the *emlSGX-PM* node. As previously discussed, the performance improvement obtained with PLINIUS's PM-based save-restore mechanism PM can be attributed to the short I/O path from the application's address space to PM, as opposed to the longer I/O path with SSD where data goes through the page cache before it is eventually written to slower secondary storage, *i.e.*, SSD.

Nonetheless, it is worth mentioning that the performance improvement provided by PM on the *emlSGX-PM* node is higher than what would be obtained with real PM hardware, as DRAM exhibits lower latency compared to PM.

**CPU and memory overhead.** Our mirroring mechanism uses 140 bytes of PM for encryption metadata per layer. The MAC is 16 B, the IV is 12 B, giving 28 B per encrypted parameter buffer. Each layer contains 5 parameter matrices, hence $28 \times 5 = 140$ B per layer. With a model of $N$ layers, we account for $N \times 140$ extra bytes on PM for encryption metadata, small compared to the size of actual models (order of few MBs). The training algorithm is a fairly intensive single-threaded application and it uses 98-100% of the CPU during execution.

**Training larger models.** Our results suggest PLINIUS is best suited for models with sizes beneath the EPC limit. Models larger than the EPC limit can be trained with PLINIUS but this leads to a significant drop in training performance due to the extensive page swaps by the SGX kernel driver. Figure 6.7 shows our mirroring mechanism still peforms better than SSD-based checkpointing for model sizes beyond the EPC limit. A possible strategy to overcome the EPC limitation could be to distribute the training job over multiple secure CPUs. This idea will be explored in the future. Recent Intel Xeon Scalable processors support up to 1 TB of EPC, 512 MB per CPU [71]. This paves the way for applications that leverage PLINIUS to train larger models more efficiently.

**Mirroring frequency.** By default PLINIUS does mirroring after every iteration. The mirroring frequency can be easily increased or decreased. All things being equal, a training environment with a small or high frequency of failures will require respectively, small or high mirroring frequencies to achieve good fault tolerance guarantees.

### *6.8.3 Crash resilience*

*Q3: how robust is the mirroring mechanism against crashes?*

The objective of the experiments here is to demonstrate that PLINIUS's mirroring mechanism is *crash resilient* (or failure transparent), and assess the performance impact on the training process of a *non-crash-resilient* system. We define a crash-resilient system as one capable of recovering its state (*i.e.*, learned parameters) following a system crash. The experiments consider models with 5 LReLU-convolutional layers, trained with the MNIST dataset for 500 iterations. We study the variation of the loss while doing random crashes during model training.

Figure 6.8 presents the results obtained on the *emlSGX-PM* server, but similar results are obtained on *SGX-emlPM*. We proceed by training a model using PLINIUS with 9 random crashes (and resumptions) during the training process. We compare the loss curve obtained here to one obtained without any crashes (baseline). Figure 6.8(a) shows that despite the crashes, the loss curve follows closely (no breaks at crash and resume points) the one obtained without crashes. This indicates the model parameters are saved and restored correctly using the mirroring mechanism in PLINIUS.

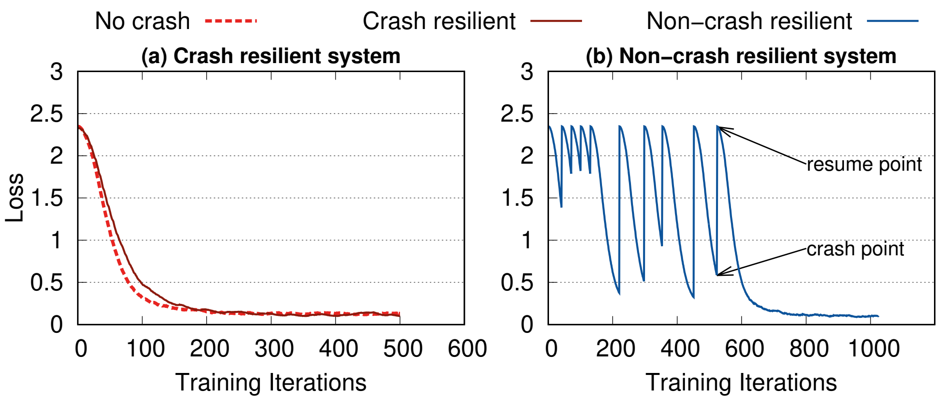


**Figure 6.8:** Crash/resumes are done by randomly killing and restarting the training process every 10 to 15 minutes during model training.

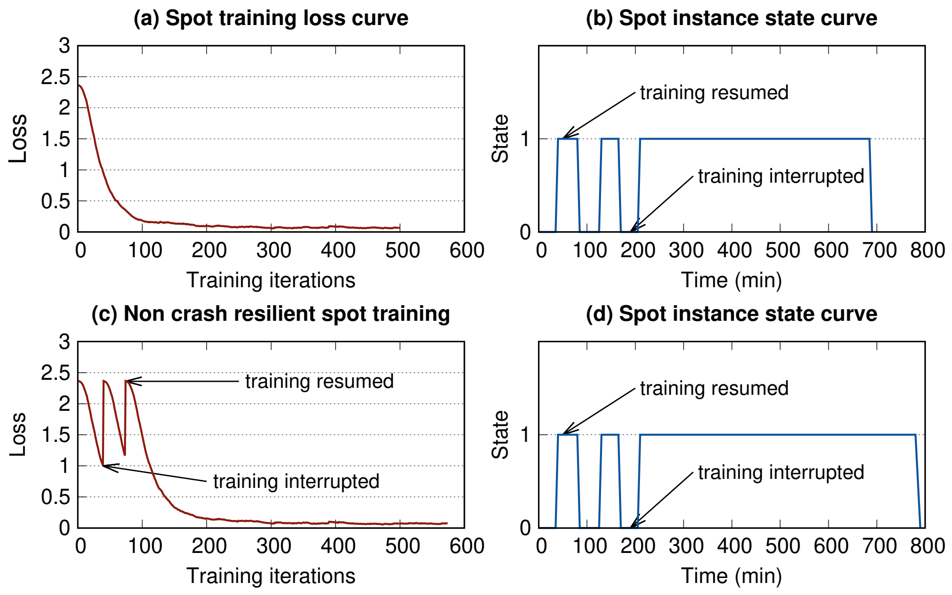


**Figure 6.9:** Model training with AWS EC2 spot instance traces.

On the contrary, Figure 6.8(b) shows what happens when we disable our mirroring mechanism. The loss curve obtained indicates the system cannot recover its learned parameters following random crashes. At every resumption point, the model begins the learning process with initial randomized weights, and thus still requires 500 iterations to be fully trained, hence increasing the total number of iterations (from when training first began) required to train the model to over 1000 in this experiment. This shows the benefit of crash-resilience in an ML system. In the next section, we use a more realistic crash/resume pattern (spot instance trace) to show crash resilience in PLINIUS.

**PLINIUS on AWS EC2 Spot instances.** A practical use case for PLINIUS framework would be model training on spot instances [219], such as those offered by Amazon EC2 and Microsoft Azure. Spot instances are liable to many interruptions during their lifetimes, and model training in such a scenario requires efficient fault tolerance guarantees (such as those provided by PLINIUS) to reduce cost and increase efficiency of the training process.

We use real Amazon EC2 spot instance traces from [205] to simulate a realistic model training scenario with PLINIUS on a spot instance.[7] The spot traces contain market prices of spot instances at different

7 Spot simulator

timestamps (5 minutes intervals). A client obtains a spot instance (or maintains a running one) if their bid price at a specific time is greater than the market price. To simulate spot model training, we set a *bid price* in our simulator script, and the simulation algorithm periodically (every 5 minutes) compares the *market price* at each timestamp in the spot trace to our bid price. If $bid\_price > market\_price$, our training process is launched (or continues if it was already running). Otherwise, the training process is killed. We train a model with 12 LReLU-convolutional layers for 500 iterations on server *emlSGX-PM*.

Figure 6.9(a) shows the loss curve obtained after 500 iterations. As explained in the previous section, this shows PLINIUS is crash resilient as training resumes where it left off prior to the training process being stopped. Figure 6.9(b) shows a "state curve" of the or spot instance throughout the training process. The state is 1 when the spont instance is running and 0 otherwise. We observe only 2 interruptions of the training process with our simulation parameters (*i.e.*, maximum bid price of 0.0955). The chosen maximum bid price and spot market price variations will dictate the total number of interruptions of the spot instance, and hence the total training time (interruption times included). The spot traces used and our simulation scripts are available in the PLINIUS repository.

Figure 6.9(c) shows us the loss curve obtained when there is no crash resilience (*i.e.*, the model's state is not saved). With the given simulation parameters (*i.e.*, maximum bid price of 0.0955), there are two interruptions during the training process. As explained in the previous section, it needs to resume training afresh, and hence the combined number of iterations (and total time) from when training first began is increased when compared to its crash resilient counterpart. This further justifies the need for fault tolerance guarantees in such ML scenarios. A system like PLINIUS has the potential to achieve these fault tolerance guarantees at a lower cost compared to regular SSD checkpointing.

**Secure inference.** PLINIUS can also be used for secure inference. We trained a CNN model with 12 LReLU convolutional layers on the MNIST training dataset, and used the trained model to classify 10,000 grayscale images of handwritten digits in the range $[0-9]$. The model (available in the PLINIUS repository) achieved an accuracy of 98.52% with the given hyper-parameters.

**GPU and TPU support.** Hardware accelerators like Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) are increasingly used in ML applications. However (at the time of this writing), the former do not integrate TEE capabilities. Recent works like HIX [75], Graviton [203], and Slalom [189] propose techniques to securely offload expensive ML computations to GPUs. Using Darknet's CUDA extensions, PLINIUS can leverage such techniques to improve training performance. The trained model weights can be securely copied between the secure CPU and the GPU (or TPU) and our mirroring mechanism applied without much changes.

## 6.9 RELATED WORK

We classify related work into 3 categories: *(i)* TEE-based ML schemes, *(ii)* Homomorphic encryption (HE)-based schemes for ML, and *(iii)* Fault tolerance techniques in ML.

**TEE-based schemes.** There exists several solutions leveraging trusted hardware (*i.e.*, Intel SGX) for secure ML. Slalom [189] is a framework for secure DNN inference in TEEs. It outsources costly neural network operations to a faster, but untrusted GPU during inference. Occlumency [95] leverages Intel SGX to preserve confidentiality and integrity of user-data during deep learning inference in untrusted cloud infrastructure. Privado[59] implements a secure inference-as-a-service, by eliminating input-dependent access patterns from ML code, hence reducing data leakage risks in the enclave. Chiron [68] leverages Intel SGX for secure ML-as-a-service which prevents disclosure of both data and code.

These systems leverage TEEs for model inference, but without any support for failure recovery. PLINIUS provides a full framework that supports both in-enclave model training and inference with efficient fault tolerance guarantees on PM.

SecureTF [164] integrates TensorFlow ML library for model training and inference in secure SCONE containers. This requires the full TensorFlow library (over 2.5 million LoC [186]) to run inside SGX enclaves, which by design increases the TCB by a lot. On the other hand, the trusted portion of PLINIUS comprises 15,900 LoC. The reduction in TCB in PLINIUS when compared to SecureTF is quite obvious; this is better from a security perspective.

**Homomorphic encryption (HE)-based schemes.** Without trusted hardware enclaves, many privacy-preserving ML methods achieve security via HE-based techniques. HE schemes compute directly over encrypted data. Some works like CryptoNets [50] implement inference over encrypted data for pre-trained neural networks. Others like [66] leverage HE to train and do inference on neural network models in a privacy-preserving manner.

While these methods ensure privacy of sensitive training and classification data during model training and inference, they have significant performance overhead (up to 1000× slower than TEE-based schemes [69]). PLINIUS provides an orthogonal approach to tackle security, combining Intel SGX enclaves to ensure confidentiality and integrity of models and data sets during training and inference at a much lower cost.

**Fault tolerance in ML.** A common approach for fault tolerance in ML learning frameworks is the use of checkpointing to save a model's state to secondary storage. The corresponding file is then used to reconstruct the model following a crash. Various frameworks (*i.e.*, Tensorflow [1], Caffe [78], Darknet [36], *etc.*) rely on this method to save a model's state. Also, these frameworks rely on secondary storage for storing training data.

However, this technique comes with significant performance overhead when compared to a PM-based system, primarily due to the high access times of secondary storage. Moreover, in the event of a system crash, the entire data sets and models must be reloaded into main memory from secondary storage. PLINIUS introduces a novel mirroring mechanism which leverages PM for fault tolerance. In the event of a crash, the model and the associated training data are readily available in byte-addressable memory (PM), mitigating the performance costs associated with using slow secondary storage to achieve fault tolerance guarantees.

## 6.10 SUMMARY

PLINIUS is the first secure ML framework to leverage Intel SGX for secure model training and PM for fault tolerance. Our novel mirroring mechanism creates encrypted mirror copies of enclave ML models in PM, and synchronises the enclave and PM copies across training iterations. PLINIUS's design leverages PM to maintain byte-addressable training data in PM, completely circumventing expensive disk I/O operations in the event of a system failure. The evaluation of PLINIUS shows that its design outperforms disk-based checkpointing while ensuring the system's robustness upon system failures. Using a real-world dataset for image recognition, we show that PLINIUS offers a practical solution to securely train ML models in TEEs integrated with PM hardware at a reasonable cost.

CHAPTER 7

# SGX Switchless Calls Made Configless

Similar to the preceding chapter, the emphasis of this chapter is on enhancing the performance of enclave programs. We focus on Intel SGX switchless calls–a technique which enables cross-enclave calls without performing costly enclave transitions. In this work, we identify drawbacks in Intel's static switchless calls approach, and propose a dynamic and configless approach, facilitating enclave software development and deployment.

The chapter is organised as follows:

## 7.1 INTRODUCTION

As discussed in the previous chapter, SGX enclaves must perform a context switch to invoke system calls like `read`, `write`, *etc*. which are ubiquitous in contemporary applications. This context switch can cost up to 14,000 CPU cycles [210], representing about 93× overhead when compared to a regular OS system call, which incurs only 150 CPU cyles. This overhead is mainly due to the CPU flushing its caches as well as all TLB entries containing enclave addresses, so as to preserve confidentiality when switching the CPU's state between enclave and non-enclave mode [34], and represents a serious performance bottleneck for applications which perform many enclave transitions for I/O.

Techniques have been proposed [7, 210, 157] to circumvent expensive enclave context switches by leveraging *worker threads* in and out of the enclave. *Client threads*, *i.e.*, inside or outside of the enclave, send their requests to the opposite side via shared memory. Worker threads handle such requests and send the appropriate responses once completed. The Intel SGX SDK implements this technique via the *switchless call library* [31] and recommends configuring a routine as switchless when it is *short in duration and frequently invoked*. While this approach improves the performance of enclave context switches, our practical experience revealed the following problems. First, Intel SGX switchless calls **must be manually configured** at build time and misconfigurations can lead to performance degradations and waste of CPU resources [31]. Second, manually configuring `ecalls` or `ocalls` as switchless at build time is not ideal because developers rarely know the frequency nor the duration of the calls, which are typically application and workload specific.

To mitigate these problems, we propose ZC-SWITCHLESS (*zero-config* switchless), a tool to *dynamically select switchless routines* at runtime and *configure the most appropriate number of worker threads on the fly*. This approach prevents static configuration of switchless routines and worker threads, and obviates the performance penalty of misconfigured switchless systems, while minimising CPU waste. ZC-SWITCHLESS leverages an application level scheduler that monitors runtime performance metrics (*i.e.*, number of wasted cycles or fallback calls to regular non-switchless routines, *etc*.); these are used to dynamically configure an optimal number of worker threads to fit the current workload while minimising CPU waste.

We further analysed the Intel SGX SDK implementation, and derived practical configuration tips for Intel SGX switchless calls. In addition, we tracked performance issues with Intel's SDK `memcpy` implementation, and propose an optimised version (based on Intel's recommendations) which removes a major source of performance overhead for both switchless and regular SGX transition routines.

In summary, the contributions of this work are:

1. The design and the implementation of ZC-SWITCHLESS, a configless and efficient system to drive switchless calls, released as open-source.[1]
2. An optimised implementation and evaluation for the Intel SGX SDK's `memcpy` routine.
3. An extensive experimental evaluation demonstrating the effectiveness of ZC-SWITCHLESS via micro- and macro-benchmarks.

## 7.2 SWITCHLESS OCALLS

As outlined previously, *switchless calls* provide a mechanism to do cross-enclave calls without triggering an enclave context switch. The core idea is to have *caller threads* send the requests of `ecalls`/`ocalls` into shared, untrusted buffers, from which the requests are received and processed synchronously or

1 https://gitlab.com/Yuhala/zc-switchless.git

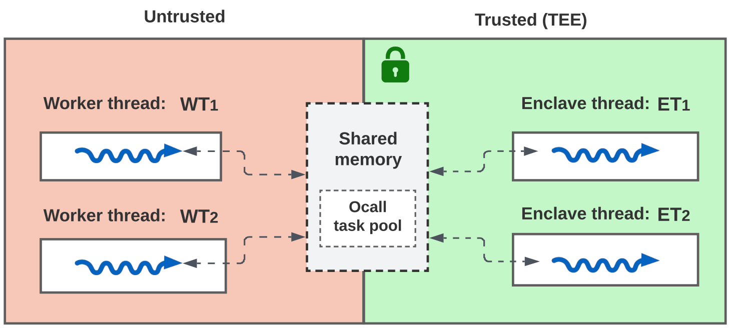


**Figure 7.1:** Intel SGX switchless `ocall` architecture.

asynchronously by worker threads in or out of the enclave [187]. In this work, we focus on `ocalls` because they are the primary source of enclave transitions from experience.

Figure 7.1 outlines the switchless `ocall` architecture in the Intel SGX SDK. *Client threads* inside the enclave send `ocall` requests to a task pool in shared memory. *Worker threads* outside the enclave wait for pending tasks in the task pool, and execute them on behalf of the client threads, returning the results in shared memory until the task pool is empty [31]. The worker threads outside the enclave perform the unsupported functionality (*e.g.*, syscalls) on behalf of in-enclave client threads. These operations are done without performing any enclave transition. The Intel SGX SDK provides a library for handling switchless calls in applications.

### 7.2.1 Limitations of Intel SGX switchless calls

In the following, we identify three main problems and limitations with Intel's switchless calls implementation: *(i)* switchless call selection (§7.2.1.1), *(ii)* worker thread pool sizing (§7.2.1.2), and *(iii)* Intel SDK parameterisation (§7.2.1.3).

**Setup.** For experiments in this section, we use a 4-core (8 hyper-threads) Intel Xeon CPU E3-1275 v6 clocked at 3.8 GHz, 8 MB L3 cache, 16 GB of RAM, with support for Intel SGX v1. We deploy Linux Ubuntu 18.04.5 LTS with kernel 4.15.0-151 and Intel SGX SDK v2.14. We report the average over 10 runs.

#### 7.2.1.1 *Switchless calls selection*

At build time, developers need to specify the routines (`ecalls` or `ocalls`) to be handled switchlessly at runtime. The Intel SGX developer reference [31] recommends configuring a routine as switchless if it is *short* in duration and is *frequently called*. However, obtaining such details at build time is challenging for developers. To illustrate this challenge, consider a program implemented in Java but executed as a native image in an SGX enclave (Chapter 3). Predicting the specific `libc` routines invoked by the in-enclave shim library could be difficult for a non-expert developer. Moreover, auto-tuning tools [110] cannot easily spot potential switchless routines by relying solely on duration and frequency, as it would require a full code path exploration, a costly operation for large systems. Additionally, the execution frequency of a specific application routine is workload specific and hard to predict at build time.

The lack of sufficient information at build time may result in the misconfiguration of switchless routines, potentially impacting the application's performance. To illustrate this, we use a synthetic benchmark. It executes $n$ `ocalls` to two functions as follows: $\alpha$ calls to function $f$ which is known to benefit from switchless calls (*i.e.*, short in duration [187]), while $\beta$ calls to $g$ which should run as a regular `ocall`. If $\alpha = \beta$, the sum of the execution times of calls to $f$ is negligible compared to the same sum for calls

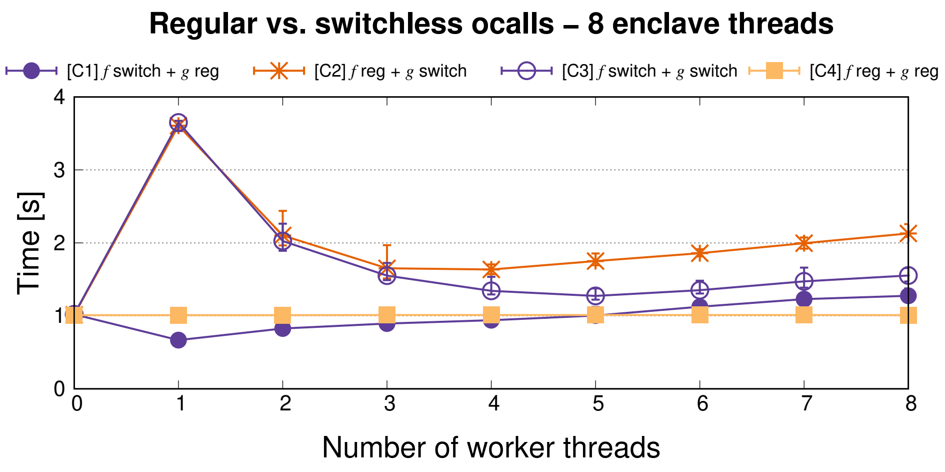


**Figure 7.2:** Runtime for 75,000 switchless `ocalls` to $f$ and 25,000 regular `ocalls` to $g$

to $g$. Hence, to better highlight the performance gains when executing $f$ switchlessly, we set $\alpha = 3\beta$ and $n = \alpha + \beta$. In this test, $f$ is an empty function (*i.e.*, `void f(void){}`). On the other hand, $g$ routine executes `asm("pause")` in a loop, *i.e.*, a busy-wait loop.

We evaluate four different configurations: `C1`-`C4`. In `C1`, all $f$ functions run switchlessly, while $g$ functions run as regular `ocalls`. We expect `C1` to perform best. In `C2`, only the $g$ functions run switchlessly, and we expect `C2` to be the worst. Finally, in `C3` and `C4` all functions run switchlessly or regularly, respectively. We observe the following results (on average) when executing 100,000 `ocalls`, *i.e.*, switchless and regular `ocalls` combined. `C1` is the fastest configuration (0.9 s), $\approx 1.8\times$ faster than `C2`, which is indeed the worst configuration (completes in 1.6 s). `C4` completes in 1.3 s ($1.44\times$ slower than `C1`).

**Take-away 1**

An improper selection of switchless coroutines leads to poor performance in SGX applications.

### 7.2.1.2 *Worker thread pool sizing*

In addition to switchless routine choice, developers must also specify the total number of worker threads allowed to the SGX switchless call mechanism at build time. However, an overestimation of worker threads for `ocalls` can lead to a waste of CPU resources due to busy-waiting of worker threads awaiting switchless requests (see [31], page 71). The latter will limit the number of applications that can be co-located on the same server or interfere with application threads which will be deprived of CPU resources. Similarly, an underestimation can lead to poor application performance as more `ocalls` will perform costly enclave switches.

Using the same synthetic benchmark from §7.2.1.1, we validate these effects for a varying number of worker threads. The results are depicted in Figure 7.3. Executing all functions switchlessly (`C4`) is good for short $g$ functions, scaling with the available worker threads. As a general trend, the performance of all configurations plateaus from 4 to 5 worker threads. In other words, statically configuring 4 or 5 worker threads for the application leads to a waste of CPU resources as optimal performance is attained with 1 to 3 worker threads.

**Take-away 2**

An overestimation of worker threads leads to a waste of CPU resources.

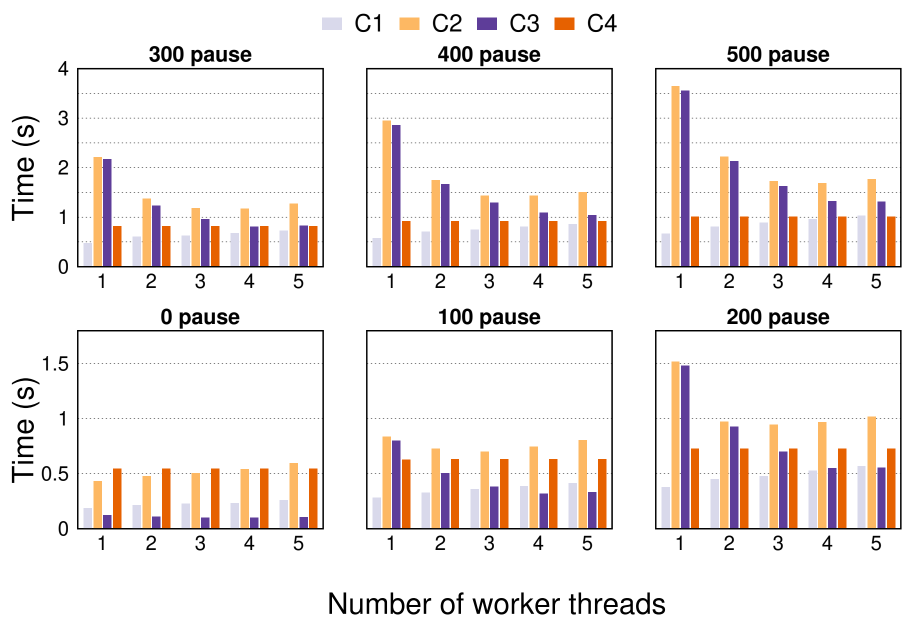


**Figure 7.3:** Runtime for 100K `ocalls` with 8 in-enclave threads for different durations of $g$.

```
int rbf = 20000; // number of retries before fallback to a regular ocall
bool worker_available = false;
while (!worker_available && rbf > 0) { // busy wait loop
  asm_pause();
  worker_available = find_available_worker();
  rbf--;
}
if(worker_available) { do_switchless_ocall(); }
else { do_normal_ocall(); } // fallback to a regular ocall
```

**Listing 15:** Illustrating retry before fallback

### 7.2.1.3 *Intel SDK parameterisation*

If a switchless task pool is full or all worker threads are busy, a switchless call falls back to a regular `ecall` or `ocall`. The Intel SGX SDK defines the variable `retries_before_fallback` (`rbf`) for the number of retries client threads perform in a busy-wait loop, waiting for a worker thread to start executing a switchless call request, before falling back to a regular `ecall`/`ocall` [31]. Similarly, the SDK defines `retries_before_sleep` (`rbs`), for the number of `asm{"pause"}` done by a worker while waiting for a switchless request, before going to sleep. The SDK's default values for both `rbf` and `rbs` are set at 20,000 retries.

However, the SDK's default `rbf` value especially is abnormal in both a theoretical and practical sense. Between successive retries, a caller thread executes an `asm{"pause"}` instruction, which has an estimated latency up to 140 cycles on Skylake-based microarchitectures.[2] As a result, a caller thread can wait more than 2.8 million cycles before its call is handled by a worker thread. This is about 200× more costly compared to a regular `ocall` transition (≈14,000 cycles), and defeats the purpose of using switchless calls, *i.e.*, to avoid the expensive transition. The same holds for `rbs`; a worker thread will wait for 2.8 million cycles before going to sleep.

While developers can easily tune the `rbf` and `rbs` values at build time, the Intel SDK lacks proper guidance.

2 SGX was first introduced on Skylake CPUs: `https://intel.ly/3hTVEMG`, page 58

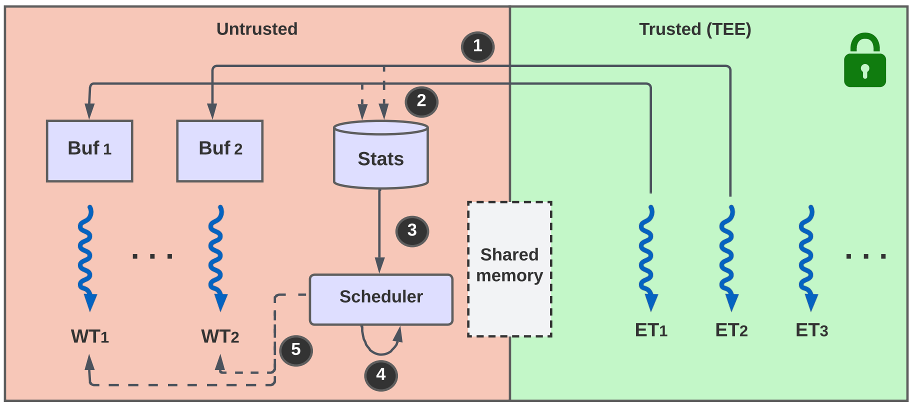


**Figure 7.4:** ZC-SWITCHLESS general overview.

**Take-away 3**

The proper configuration of the Intel SGX switchless-related parameters remains hard and misconfigurations lead to poor performance.

## 7.3 ZC-SWITCHLESS ARCHITECTURE

To address the aforementioned limitations, we propose ZC-SWITCHLESS, a dynamic approach to drive SGX switchless calls more effectively. The primary objective of ZC-SWITCHLESS is to provide a resource-efficient implementation for SGX switchless calls.

Central to ZC-SWITCHLESS's architecture is an in-application scheduler (§7.3.1) which aims to minimise waste of CPU resources. Additionally, we detail a more efficient implementation of `tlibc`'s `memcpy` (§7.3.5), used for intra-enclave data copying, as well as data exchange between the enclave and the outside world.

Figure 7.4 shows an overview of ZC-SWITCHLESS. In a nutshell, we consider *any function* (Figure 7.4-❶) as a potential candidate to run as switchless, thus avoiding the need for manual selection by developers at build time. Our design allows the number of worker threads to be tuned dynamically, to minimise CPU waste while improving performance via the switchless call technique. We do so by having the scheduler implement a feedback loop to periodically collect `ocall` statistics (Figure 7.4:❷-❸), determine the optimal number of worker threads (Figure 7.4-❹), and finally apply its decision (Figure 7.4-❺). Inside the enclave, a call is executed in a switchless manner if the caller finds at least one idle worker thread. The remainder details further these aspects. Unless indicated otherwise, the term *scheduler* refers to ZC-SWITCHLESS's scheduler.

### *7.3.1 ZC-SWITCHLESS scheduler*

The main objective of the scheduler is to minimise wasted CPU cycles. We define a *wasted CPU cycle* as one spent by a CPU core doing something that does not make the application (*i.e.*, caller thread) move forward in its execution [108].

In the case of Intel SGX, we identify two potential sources of wasted CPU cycles: *(1)* transitions between enclave and non-enclave mode in the case of regular `ocall`s, and *(2)* busy-waiting in the case of switchless `ocall`s. The overhead of regular `ocall`s has been evaluated extensively in past research [210], and it varies also according to the specific CPU and micro-code version. We evaluated

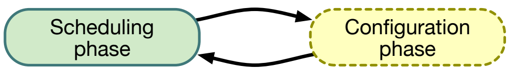


**Figure 7.5:** ZC-SWITCHLESS scheduler phases.

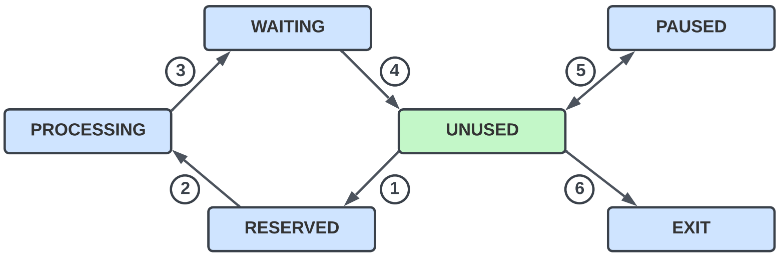


**Figure 7.6:** Worker thread state transitions.

this overhead to be $\approx 13,500$ CPU cycles for our experimental setup.

An any point in time, each *active worker* (not sleeping) can be in one of two states: either the worker is handling an `ocall`, in which case the enclave thread (which made the `ocall`) is busy-waiting, or the worker is busy-waiting for incoming `ocall` requests. Therefore, for every active worker thread, there is always exactly one thread busy-waiting. The extra cost of having $M$ worker threads is thus $M$ multiplied by the number of cycles during which they have been active.

The scheduler periodically computes the number of worker threads that minimises the number of wasted CPU cycles. Throughout its lifetime, the scheduler switches between two phases (see Figure 7.5): a *scheduling phase* that lasts a scheduler quantum, $Q$, a time slice during which it assigns an optimal number of switchless workers to be used by the application, and a *configuration* phase, during which it calculates the optimal number of switchless workers for the next scheduling phase. We denote by $T_{es}$ the duration of an enclave switch, $F$ the number of calls not being handled switchlessly (*i.e.*, fallback), $N$ the number of cores on the machine and $M$ the number of worker threads set during ZC-SWITCHLESS's scheduler quantum ($Q$, set empirically to 10 ms). The number of wasted cycles during $T$ cycles is given by:

$$U = F \cdot T_{es} + M \cdot T$$

This formula takes into consideration wasted cycles due to fallback routines: $F_i \cdot T_{es}$ and the $M$ busy-waiting threads: $M \cdot T$, *i.e.*, enclave threads waiting for a response from their switchless worker threads, or worker threads busy-waiting for a switchless request from enclave threads. Finally, the scheduler keeps $M'$ workers for the next *scheduling phase*, where $M'$ is such that $U_{M'} = \min_i U_i$.

To estimate the number of workers for the next quantum (*i.e.*, *scheduling phase*) during a *configuration phase*, the scheduler sleeps for $\frac{N}{2} + 1$ micro-quanta, of length $\mu \cdot Q$ each, with a different number of workers $i$ each time, $0 \leq i \leq \frac{N}{2}$ (*i.e.*, $\frac{N}{2} + 1$ possible values). The constant $\mu$ is a small time period, so the configuration phase can be quick, but still long enough to capture the needs (in terms of CPU resources) of the application at a given time. We empirically set $\mu = \frac{1}{100}$. During every micro-quantum, the number of non-switchless calls is recorded so that the scheduler can compute the number of wasted cycles $U_i$ once awake. $U_i$ is given by:

$$U_i = F_i \cdot T_{es} + i \cdot \mu \cdot Q \cdot CPU\_FREQ$$

To deactivate a worker thread, the scheduler sets a value in memory (see §7.3.2). The worker's loop function will eventually check this value and, if it is set and no caller thread has reserved (or is using) the worker, the worker will pause. To re-activate a paused worker, the scheduler sends a signal to wake up the corresponding worker thread.

### 7.3.2 *Worker thread state machine*

We associate to each worker a `buffer` structure that consists of 4 main fields: an untrusted (preallocated) memory pool used by callers to allocate switchless requests, a field to hold the most recent switchless request, a status field to track the worker status, and a field used to communicate with the scheduler. Figure 7.6 summarises the `status` transitions of a worker thread.

A worker is initially in the `UNUSED` state. When a caller needs to make a switchless call, it finds an `UNUSED` worker and switches the worker's state to `RESERVED` (①). We rely on GCC atomic built-in operations [52] for thread synchronisation. The caller allocates a switchless request structure from the corresponding memory pool. A switchless request comprises: an identifier of the function to be called, the function arguments (if present), and the return result (if present).

The caller copies its request to the worker's buffer and changes the worker's state from `RESERVED` to `PROCESSING` (②). At this point, the worker reads the request and calls the desired function with the corresponding arguments. Once the function call completes, the worker updates the request with the returned results (if present) and switches from the `PROCESSING` to the `WAITING` state (③).

Finally, the caller copies the returned results into enclave memory and changes the worker's state to `UNUSED` (④). After a scheduling phase, the number of additional worker threads ($\frac{N}{2} - M'$) are paused by the scheduler (⑤).

The memory pools of worker buffers are freed and re-allocated when full via an ocall. Using preallocated memory pools prevents callers from performing `ocalls` to allocate untrusted memory for each switchless request, which will defeat the purpose of using a switchless system.

Upon program termination, the scheduler sets a value in workers' buffers so the workers can switch to the `EXIT` state (⑥). At this stage, the workers perform final cleanup operations (*e.g.*, freeing memory) and then terminate.

### 7.3.3 *Switchless call selection*

In ZC-SWITCHLESS, any routine can be run as switchless if the corresponding enclave caller thread finds an available/unused worker thread. Otherwise, the call immediately falls back to a regular `ocall` without any busy waiting.

### 7.3.4 *Security analysis of* ZC-SWITCHLESS

The security analysis of ZC-SWITCHLESS is similar to that presented in [210]. All switchless call designs (*i.e.*, Intel switchless, ZC-SWITCHLESS, Eleos [157]) are based on threads in and out of the enclave communicating via plaintext shared memory. Thus, the switchless design proposed by ZC-SWITCHLESS is no less secure than that proposed by the Intel SGX switchless library. In the following, we do a more precise security analysis of key components in ZC-SWITCHLESS's design.

**ZC switchless request structure.** It is a `C struct` which encapsulates all the data necessary for a switchless request: an identifier for the function to be called, the arguments passed (if any), the status of the switchless request, and the results. A potential security risk could be Iago attacks [19], *i.e.*, maliciously modifying return values so as to manipulate the control flow of the enclave. Similar to [10], ZC-SWITCHLESS mitigates this by validating inputs and return values at the enclave boundary, *e.g.*, ensuring that the return value of a libc call is consistent with its specification.

**ZC scheduler.** The ZC-SWITCHLESS scheduler is located in the untrusted runtime, which leaves it vulnerable to malicious tampering. However, since the scheduler only decides how many worker

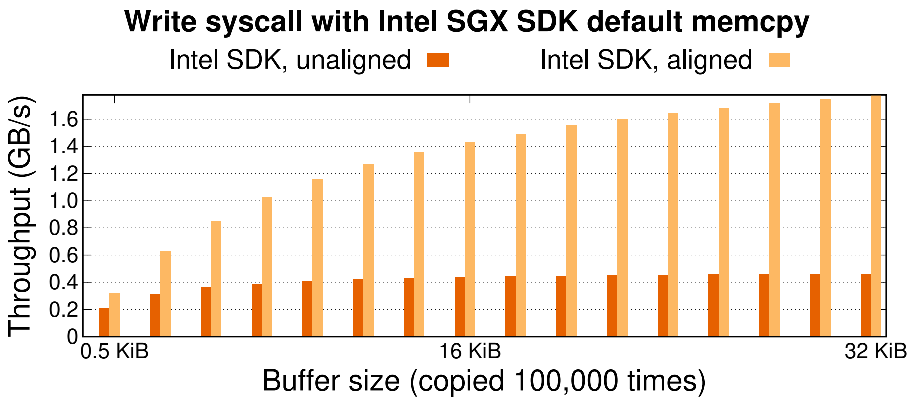


**Figure 7.7:** Throughput for `ocalls` of the `write` system call to `/dev/null` (100,000 operations, average over 10 executions) for aligned and unaligned buffers.

threads should be used and when, the worst case scenario here will be a DoS, *e.g.*, by killing worker threads. The enclave's confidentiality or integrity, however, cannot be compromised by tampering with the scheduler.

### *7.3.5 Trusted libc's* `memcpy` *optimisation*

For security reasons, the Intel SDK provides its own implementation of a subset of the functions from the `libc`, *i.e.*, the `tlibc`. The `tlibc` re-implements most of the functions from the `libc` that do not require system calls, *e.g.*, `memset`, `memcpy`, `snprintf`, *etc*. Because enclave code cannot be linked to dynamic libraries, these re-implementations are statically linked to the enclave application at build time.

We focus on the Intel SDK `tlibc` version of `memcpy` [172], as it is heavily used to pass `ocall` arguments from trusted memory to untrusted memory and back for the results [98].

Our tests highlight huge performance gaps when using aligned buffers (*i.e.*, when the `src` and `dest` arguments are a multiple of 8 ) and unaligned buffers.

For instance, Figure 7.7 presents the throughput when issuing 100,000 invocation of the `write` system call (which requires an `ocall` and involves `memcpy` calls) with varying lengths of aligned and unaligned buffers ranging from 512 B to 32 KB. We observe that the execution time for unaligned buffers is consistently higher than for aligned buffers. Moreover, when using unaligned buffers, we observe poor scalability trends for `write` when increasing the buffer sizes, basically plateauing at about 0.4 GB/s.

Intel's software optimisation reference[3] provides recommendations for optimising operations like `memcpy` on various processor architectures, beginning with those based on the Ivy Bridge microarchitecture. These optimisations include leveraging instructions like enhanced `rep movsb`[4] and `stosb`[5] for buffer copy operations.

By analysing the Intel SGX SDK's implementation of `tlibc`'s `memcpy`, we observed such recommendations are not always put into practice. For instance, the SGX SDK performs a *software word-by-word copy for aligned buffers* but *a byte-by-byte copy for unaligned buffers*.

3 Intel 64 and IA-32 Architectures Optimization Reference Manual Volume 1
4 `movsb`: "move string byte", fetches a byte from a source address and stores it at a destination address.
5 `stosb`: "store string byte", stores a byte at a destination address.

```
void *memcpy(void *dst0, const void *src0, size_t length){
    ...
    /* Copy forward. */
    __asm__ volatile(
        "rep movsb"
        :   "=D"(dst0), "=S"(src0), "=c"(length)
        :   "0"(dst0), "1"(src0), "2"(length)
        :   "memory");
done:
    return (dst0);
}
```

**Listing 16:** Optimising `memcpy` with `rep movsb`.

By following those recommendations, we provide a revised and more efficient implementation leveraging the hardware instruction `rep movsb`. Listing 16 sketches the optimised `memcpy` implementation, which we evaluate in depth in §7.4.4.

Similar optimisation techniques can be employed for operations like `memmove`, `memset`, *etc*.

## 7.4 EVALUATION

Our evaluation seeks to answer the following questions:

**Q1**: How does ZC-SWITCHLESS impact application performance for (i) static (§7.4.1.1), and (ii) dynamic (§7.4.3.1) workloads?
**Q2**: What is the effect of misconfigurations of Intel switchless on application performance? (§7.4.1.1) and (§7.4.3.1)
**Q3**: What is the effect of ZC-SWITCHLESS on CPU utilisation for static (§7.4.1.2) and dynamic (§7.4.3.2) workloads?
**Q4**: What is the performance gain of the improved `memcpy` implementation? (§7.4.4)

**Experimental setup.** All experiments use a server equipped with a 4-core Intel Xeon CPU E3-1275 v6 clocked at 3.8 GHz with hyperthreading enabled. The CPU supports Intel SGX, and ships with 32 KB L1i and L1d caches, 256 KB L2 cache and 8 MB L3 cache. The server has 16 GB of memory and runs Ubuntu 18.04 LTS with Linux kernel version 4.15.0-151. We run the Intel SGX platform software, SDK, and driver version v2.14. All our enclaves have maximum heap sizes of 1 GB. The EPC size is 128 MB (93.5 MB usable by enclaves). We use both static and dynamic benchmarks.

Our static benchmarks are based on `kissdb` [70] and an Intel SGX port of `OpenSSL` [32]: `kissdb` is a simple key/value store implemented in plain C without any external dependencies, while `OpenSSL` [47] is an open-source software library for general-purpose cryptography and secure communication.

For dynamic benchmarks, we use `lmbench` [113], a suite of simple, portable, ANSI/C microbenchmarks for UNIX/POSIX.

For all benchmarks, we set the initial number of worker threads to $\frac{\#logical\_cpus}{2}$ for ZC-SWITCHLESS. This number is 4 for the SGX server used. For Intel switchless experiments, we maintain the default `rbf` and `rbs` values, *i.e.*, $20,000$.

### 7.4.1 *Static benchmark:* `kissdb`

We issue a varying number of key/value pair writes to `kissdb`, and we evaluate and compare the performance and CPU utilisation of ZC-SWITCHLESS with Intel switchless.

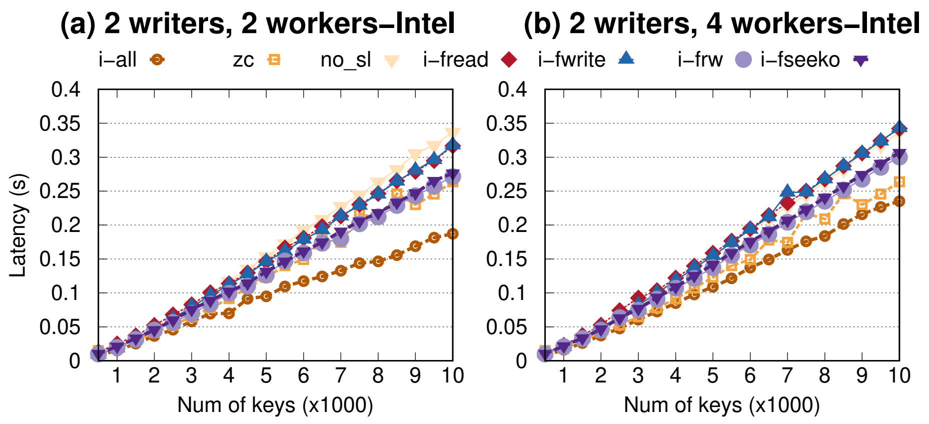


**Figure 7.8:** Average latency of key/value pair SET ops in kissdb.

We ran our benchmark in 3 modes: without using switchless calls (`no_sl`), using Intel switchless calls, and using ZC-SWITCHLESS (`zc` for short). For Intel switchless, we consider two values for the number of switchless worker threads: 2 and 4. In `kissdb`, we know empirically that the 3 most frequent `ocalls` in the benchmarks are: `fseeko`, `fwrite`, and `fread`. Therefore, for `kissdb` we benchmark Intel's switchless in 10 ($2 \times 5$) different configurations: only `fseeko` as switchless (`i-fseeko`-$x$, $x$ being 2 or 4, the number of Intel switchless worker threads), only `fwrite` as switchless (`i-fwrite`-$x$), only `fread` as switchless (`i-fread`-$x$), both `fread` and `fwrite` as switchless (`i-frw`-$x$), and all the 3 `ocalls` as switchless (`i-all`-$x$). Note that these ten configurations correspond to possible configurations an SGX developer could have set up for `kissdb`. We report the averages over 5 runs.

#### 7.4.1.1 ZC-SWITCHLESS vs. *Intel switchless*

*Answer to Q1 & Q2.* Figure 7.8 (a) and (b) show the average latencies for setting a varying number of 8-byte keys and 8-byte values in `kissdb`, with respectively 2 and 4 switchless worker threads configured for Intel switchless.

*Observation.* Here, ZC-SWITCHLESS is 1.22× faster when compared to a system without switchless calls, and respectively 1.19×, 1.13×, 1.13×, 1.05×, and 1.02× faster when compared to `i-fread`-2, `i-fwrite`-2, `i-fseeko`-2, and `i-frw`-2.

However, ZC-SWITCHLESS is about 1.33× slower for `i-all`-2. We note how ZC-SWITCHLESS is (on average) 1.26× faster than `i-fread`-4, 1.22× faster than `i-fwrite`-4, 1.13× faster than `i-fseeko`-4, 1.10× than `i-frw`-4. However, it is 1.16× slower than `i-all`-4.

*Discussion.* In `kissdb`, `fseeko` is the most frequent `ocall`, invoked almost twice more often than `fread` and `fwrite`. Further, `fseeko` is much shorter in duration relative to `fread` and `fwrite`, which explains the better performance of `i-fseeko` when compared to `i-fread` and `i-fwrite` for both 2 and 4 switchless worker threads. `i-fwrite` configurations show the poorest performance for Intel's switchless in all cases.

However, when both `fread` and `fwrite` are configured as switchless (`i-frw`), we see an improvement in performance relative to `i-fwrite` and `i-fread`, almost equal to `i-fseeko` performance. Here the combined sum of `fread` and `fwrite` calls surpasses the number of `fseeko` invocations, which leads to a more significant number of switchless calls in `i-frw`, thus leading to similar performance as `fseeko`.

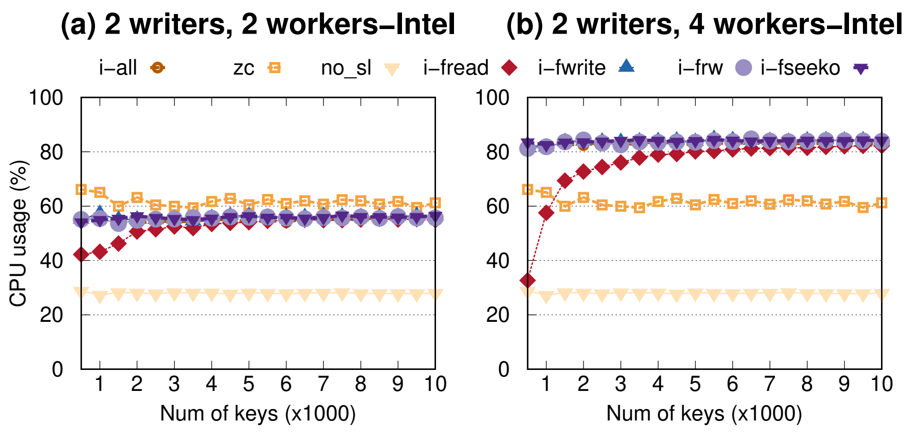


**Figure 7.9:** Average CPU usage of key/value pair SET ops in `kissdb`.

The configless strategy employed by ZC-SWITCHLESS outperforms statically misconfigured systems like `i-fread` and `i-fwrite` (for both 2 and 4 Intel switchless workers), shows similar performance as `i-fseeko`-2, and outperforms `i-fseeko`-4. The observed spikes (*e.g.*, 7,500 and 8,500 keys) in ZC-SWITCHLESS are due to `ocall` operations when reallocating full memory pools for ZC-SWITCHLESS worker buffers (see §7.3.2). In `i-all`, Intel's switchless outperforms ZC-SWITCHLESS because it maintains a constant number of switchless workers (2 or 4), whereas ZC-SWITCHLESS uses few worker threads at various points during the application's lifetime.

**Take-away 4**

ZC-SWITCHLESS achieves better performance relative to non-switchless systems, and outperforms misconfigured Intel switchless systems.

### 7.4.1.2 *ZC-switchless vs. Intel switchless: CPU usage*

*Answer to Q3.* We now evaluate the CPU utilisation of ZC-SWITCHLESS and Intel switchless when running the same experiments as in §7.4.1.1. We measured the overall CPU utilisation for the given systems from the kernel's `/proc/stat`. The percentage CPU utilisation is calculated by:

$$\%\ cpu\_used = \frac{(user + nice + system)}{(user + nice + system + idle)} * 100$$

where: *user* is the time spent for normal processes executing in user mode, *nice* is the time spent for processes executing with "nice" priority in user mode, *system* is the time spent for processes executing in kernel mode, and *idle* is time spent by the CPU executing the *system idle process*.

Figure 7.9 shows the average CPU usage for setting a varying number of 8-byte keys and 8-byte values in `kissdb`, with respectively 2 and 4 switchless worker threads configured for Intel switchless.

*Observation.* The experimental results show that ZC-SWITCHLESS maintains approximately 60% CPU usage throughout the benchmark's lifetime. For 2 Intel switchless workers, all Intel switchless configurations stabilise at about 55% CPU usage, while for 4 Intel switchless workers, the Intel switchless configurations stabilise at about 80% CPU usage. All switchless systems have visibly higher CPU usage when compared to the system without switchless calls enabled (`no_sl`).

*Discussion.* Intel's switchless mechanism maintains a constant number of worker threads (2 or 4) throughout the application's lifetime, while ZC-SWITCHLESS's scheduler increases or decreases the number of worker threads (to a maximum of 2 or 4) with respect to the workload. This explains the overall lower CPU usage of ZC-SWITCHLESS relative to Intel switchless.

**Take-away 5**

Poorly configured Intel switchless systems, *e.g.*, `i-fread`, `i-fwrite`, `i-frw` lead to a waste of CPU resources. ZC-SWITCHLESS obviates the performance penalty from misconfigured Intel switchless systems, while minimising CPU waste.

### *7.4.2 Static benchmark: OpenSSL file encryption/decryption*

This benchmark consists of two enclave threads encrypting and decrypting data read from files. The first thread reads chunks of plaintext from a file, encrypts these in the enclave, and writes the corresponding ciphertext to another file, while the second thread reads ciphertext from a different file, and decrypts it inside the enclave. All cryptographic operations are done with the AES-256-CBC [162] algorithm.

Similarly, we ran this benchmark in the 3 modes described above: `no_sl`, using Intel switchless calls (2 and 4 workers), and ZC-SWITCHLESS.

In the `OpenSSL` benchmark, we know empirically that 4 `ocalls` are called most frequently: `fread`, `fwrite`, `fopen`, and `fclose`. We consider 10 possible configurations ($2 \times 5$) for Intel's switchless routines: only `fread` as switchless (`i-fr-`$x$, $x$ being 2 or 4, the number of Intel switchless worker threads), only `fwrite` as switchless (`i-fw-`$x$), both `fread` and `fwrite` (`i-frw-`$x$), both `fopen` and `fclose` (`i-foc-`$x$), and all 4 `ocalls` as switchless (`i-frwoc-`$x$).

Figure 7.10 shows the latency and CPU usage for these configurations. We highlight and discuss essential observations.

*Observation and discussion.* In this `OpenSSL` benchmark, `fread` and `fwrite` are respectively called $\approx 2700\times$ and $\approx 1400\times$ more frequently as compared to both `fopen` and `fclose`. This explains why Intel's latency is poor (close to `no_sl`) with `i-foc`, as more ocalls perform context switches, and much better (w.r.t `no_sl`) with `i-frw`, where a larger number of ocalls are performed switchlessly. Intel performs best with `i-frwoc`, when the four ocalls are all configured as switchless. However, we observe that ZC-SWITCHLESS is about 1.62× and 1.82× faster than Intel's best configuration `i-frwoc`, for 2 and 4 Intel workers respectively. This is explained by the fact that the `fread` and `fwrite` calls here are much longer (about 6× longer as compared to the previous `kissdb` benchmark). This accentuates the bad effect of the poor default `rbf` value in Intel's switchless, as enclave threads do longer pauses waiting for a switchless worker thread to become available. This is absent in ZC-SWITCHLESS, where caller threads immediately fall back.

Regarding CPU usage, ZC-SWITCHLESS's scheduler set the number of worker threads to 0, 1, 2, 3, 4 for respectively 9.4%, 4.6%, 84.4%, 1.6%, and 0% of the program's lifetime. This explains the similar CPU usage in ZC-SWITCHLESS and Intel 2 workers (except for `i-foc`), while with 4 Intel workers, CPU usage for Intel's best config is about 1.62× larger than ZC-SWITCHLESS's, despite the latter performing better.

**Take-away 6**

ZC-SWITCHLESS outperforms all Intel configurations when `ocalls` are long; this is due to the poor default `rbf` value in Intel's switchless library.

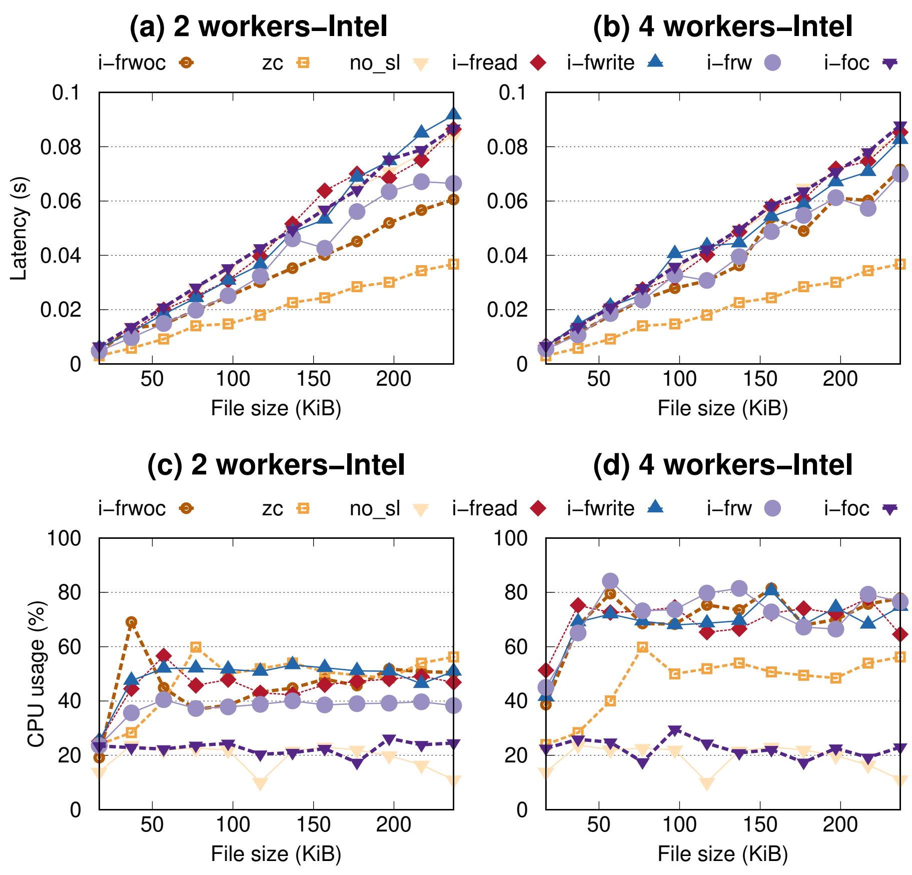


**Figure 7.10:** Latency and CPU usage for OpenSSL.

### 7.4.3 *Dynamic benchmark:* `lmbench`

Our dynamic benchmark is based on the `read` and `write` system call benchmarks of `lmbench`. The `read` benchmark iteratively reads one word from the `/dev/zero` device [113], while the `write` benchmark iteratively writes one word to the `/dev/null` device. We devised a dynamic workload approach which consists of periodically (every $\tau = 0.5s$) issuing a varied number of read and write operations to `lmbench` using two in-enclave caller threads (1 reader + 1 writer) over a period of $60s$. These operations trigger `ocalls`, and ZC-SWITCHLESS scheduler adapts the number of worker threads accordingly.

The total run time of the dynamic benchmark is divided into 3 distinct phases, each lasting $20s$: *(1) increasing operation-frequency*: the number of operations is doubled periodically, *(2) constant operation-frequency*: the number of operations remains at a constant value (the peak value from phase-1), and *(3) decreasing operation-frequency*, where the number of operations is periodically decreased (reduced by half every $\tau$). We measure the read/write throughputs and CPU usage at different points during the benchmark's lifetime.

Similarly, we ran our benchmark in 3 modes: without using switchless calls (`no_sl`), using Intel switchless calls, and using ZC-SWITCHLESS (`zc`). For Intel switchless, we consider two values for the number of switchless worker threads ($x$): 2 and 4. For `lmbench` syscall benchmark, we know empirically that the `read` and `write` syscalls are the most frequent `ocalls`. So we configure Intel's switchless in six ($3 \times 2$) different configurations: only `write` as switchless (`i-write-`$x$), only `read` as switchless (`i-read-`$x$), both `read` and `write` `ocalls` as switchless (`i-all-`$x$). Similarly, these six configurations correspond to possible configurations an SGX developer could have set for their

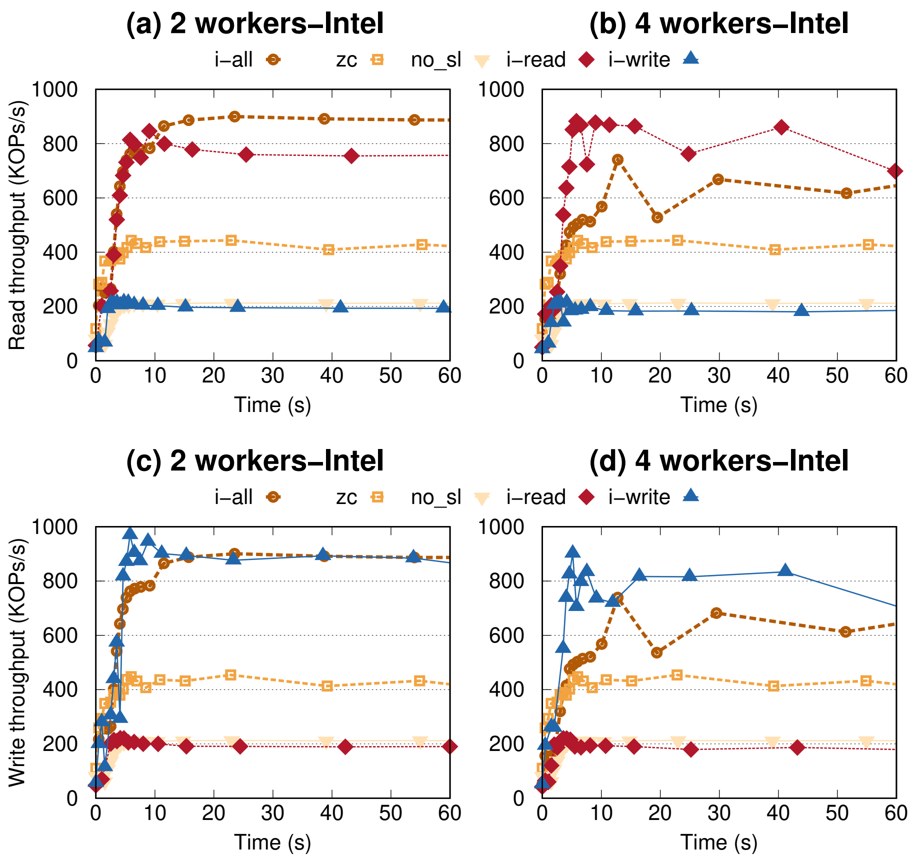


**Figure 7.11:** Read (a/b), write (c/d) throughput for ZC-SWITCHLESS vs. Intel switchless (2/4 worker threads).

program. We show the throughputs and CPU usages as observed from both the reader and writer threads. We highlight essential observations, and analyze them in our discussions.

### 7.4.3.1 *Dynamic benchmark:* ZC-SWITCHLESS *vs. Intel switchless*

*Answer to Q1 & Q2.* Figure 7.11 shows the operation throughputs as observed by the reader (top: a/b) and writer (bottom: c/d) threads respectively during the lifetime of the dynamic benchmark.

*Observation.* The experimental results show that on average, ZC-SWITCHLESS is about 2.3× faster when compared to both reader-`i-write`-2 and reader-`i-write`-4, and 2.1× and 2.5× faster when compared to writer-`i-read`-2 and writer-`i-read`-4 respectively. However, ZC-SWITCHLESS is about 1.6× and 1.1× slower when compared to properly configured Intel switchless configurations `i-all`-2 (reader,writer) and `i-all`-4 (reader, writer) respectively.

*Discussion.* From the perspective of the reader thread, `i-write` is a misconfiguration as the `read` calls (most frequently called by the reader) will never be switchless, and similarly for the writer thread, `i-read` is a misconfiguration as the `write` calls will never be switchless. This explains the relatively lower throughputs of these configurations when compared to ZC-SWITCHLESS and the other switchless configurations. However, ZC-SWITCHLESS has a lower throughput when compared to the better configurations: reader-(`i-all`,`i-read`) and writer-(`i-all`,`i-write`).

### 7.4.3.2 *ZC-switchless CPU utilisation vs. Intel switchless*

*Answer to Q3.* We compute the CPU usage as explained previously. Figure 7.12 shows the average CPU usage as observed by the reader (top) and writer (bottom) threads respectively at the different

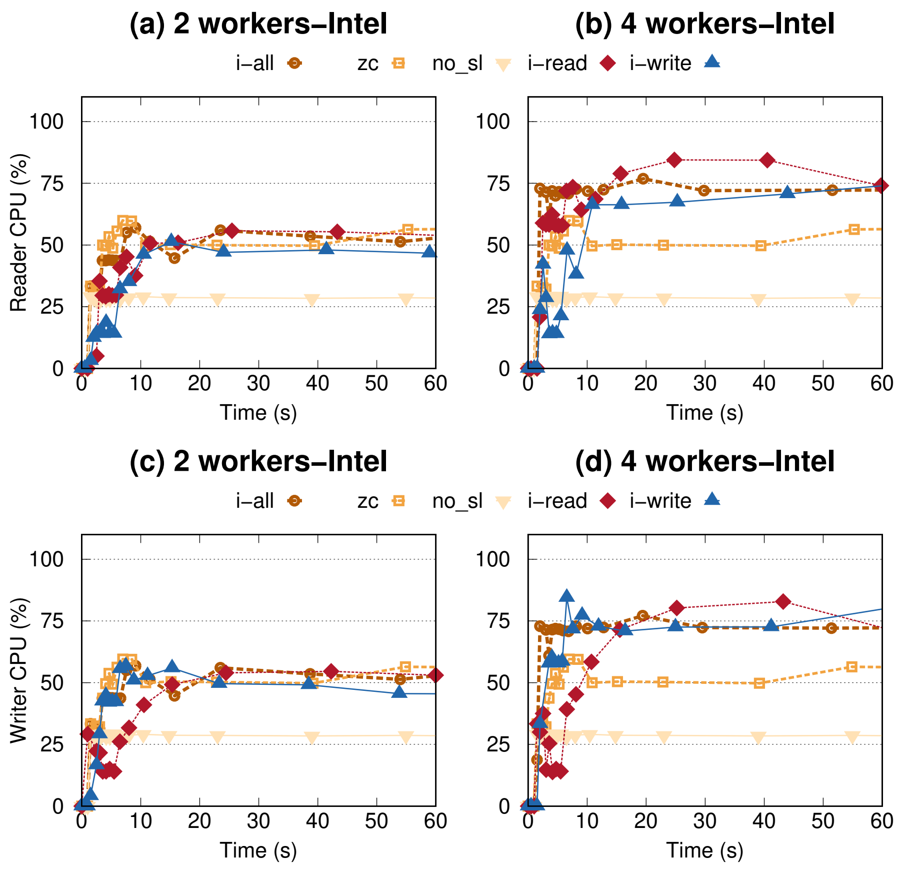


**Figure 7.12:** Read (a/b), write (c/d) CPU usage for ZC-SWITCHLESS vs. Intel switchless (2/4 worker threads).

points during the lifetime of the dynamic benchmark.

*Observation.* Similarly to the throughputs, the CPU usage for the studied configurations increases with time and plateaus at a certain point. The experimental results show that on average, ZC-SWITCHLESS CPU usage is about 1.8× and 1.6× more when compared to reader-`i-write`-2 and writer-`i-read`-2 respectively, but almost equal CPU usage on average when compared to reader-`i-write`-4, writer-`i-read`-4, and reader/writer-`i-all`-2 respectively. However, reader/writer-`i-all`-4 use about 1.3× more CPU when compared to ZC-SWITCHLESS.

*Discussion.* Similarly to the poor throughputs, we can easily highlight CPU waste for the misconfigurations: reader-`i-write`-4 and writer-`i-read`-4, as they have similar CPU usages when compared to ZC-SWITCHLESS on average but much poorer performance with regards to the corresponding throughputs. ZC-SWITCHLESS prevents this misconfiguration problem. For the better configurations: reader-(`i-read`,`i-all`), and writer-(`i-write`,`i-all`), Intel performs better than ZC-SWITCHLESS but this usually comes at a higher CPU cost (especially for 4 Intel workers), which is explained by the fact that Intel's switchless mechanism maintains the maximum number of workers when there are pending switchless requests.

The observations with the dynamic benchmark are consistent with the previous takeaways (4 and 5) indicating ZC-SWITCHLESS obviates the performance penalty of misconfigured Intel switchless systems while minimising waste of CPU resources.

### 7.4.4 *Performance of improved* `memcpy`

*Answer to Q4.* To evaluate the performance of the improved `memcpy` implementation, we

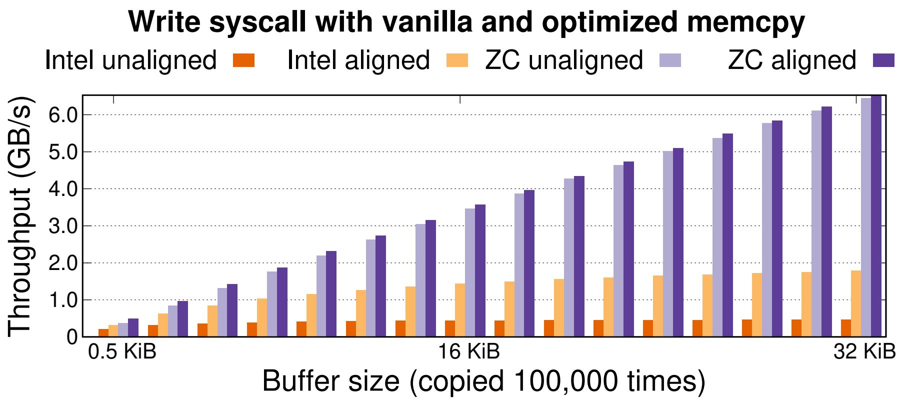


**Figure 7.13:** Throughput for `ocalls` of the `write` system call to `/dev/null` (100,000 operations, average over 10 executions) for aligned and unaligned buffers.

ran a benchmark similar to that in §7.3.5, which issues 100,000 `write` system calls from within the enclave, with various sizes of aligned and unaligned buffers ranging from 512 B to 32 KB. We ran this benchmark in two modes: using the default `memcpy` implementation of the SDK (`vanilla-memcpy`) and using our improved `memcpy` implementation (`zc-memcpy`).

As shown in Figure 7.13, the revised `memcpy` implementation achieves a speedup of up to 3.6× for aligned buffers and 15.1× for unaligned buffers. Noteworthy, these speedups will benefit both regular and switchless `ocalls`, as well as any application using the Intel SDK's `memcpy` implementation inside enclaves.

**Impact on inter-enclave communication.** Recent work [98] presents a quantitative study on the performance of real-world serverless applications protected in SGX enclaves. A performance test of `zc-memcpy` in this work showed a 7% — 15% speedup for inter-enclave SSL transfers in the context of their benchmarks, which confirms the efficiency of `zc-memcpy` for copy-intensive operations.

**Take-away 7**

Developers should have the freedom to bypass standard implementations for certain `libc` routines in favour of optimised implementations tailored to the specific microarchitectures of their underlying processors.

## 7.5 RELATED WORK

We classify related work into three categories, as detailed next.

**(1) SGX benchmarking and auto-tuning tools.** In [110], authors use stochastic optimisation to enhance the performance of applications hardened with Intel SGX. [109] proposes a framework for benchmarking SGX-enabled CPUs, micro-code revisions, SDK versions, and extensions to mitigate enclave side-channel attacks. These tools do not provide dynamic configuration of the switchless mechanisms.

**(2) SGX performance improvement.** Weichbrodt et al. [208] propose a collection of tools for high-level dynamic performance analysis of SGX enclave applications, as well as recommendations on how to improve enclave code and runtime performance, *e.g.*, by batching calls, or moving function implementations in/out of the enclave to minimise enclave transitions. *Intel VTune Profiler* [204] permits to profile enclave applications to locate performance bottlenecks, while Dingding et al. [97]

provide a framework to improve enclave creation and destruction performance. ZC-SWITCHLESS focuses on improving enclave performance efficiently via the switchless call mechanism.

**(3) SGX transitions optimizations.** Previous research works [210, 7, 157, 188] circumvent expensive SGX context switches by leveraging threads in and out of the enclave which communicate via shared memory, similar to Intel SGX's switchless design [31].

Tian et al. [187] propose a switchless worker thread scheduling algorithm aimed at maximising worker efficiency so as to maximise performance speedup. ZC-SWITCHLESS on the other hand, leverages a scheduling approach aimed at minimising CPU waste while improving application performance relative to a non-switchless system.

## 7.6 SUMMARY

In this chapter we identified drawbacks with the static configuration policy of Intel SGX's switchless library. We introduced ZC-SWITCHLESS, a system to dynamically drive switchless calls. Extensive evaluations show ZC-SWITCHLESS obviates the performance penalty from misconfigured Intel switchless systems, while minimising CPU waste.

We equally revisited some `libc` implementations, *i.e.*, `mempcy` in the Intel SGX SDK, and our experiments show significant performance improvements can be obtained with the enhanced `rep movsb` instruction. While we focused on `memcpy`, similar optimisations can be employed for routines in the same family like `memmove` and `memset`. As detailed in Intel's software optimisation manual, the performance of the proposed optimisations may vary according to the underlying processor microarchitecture. Thus, we encourage developers to have the freedom to bypass standard implementations in favour of custom implementations targetting specific processor architectures.

CHAPTER 8

# Conclusion and Outlook

Trusted execution environments represent an important technological advancement designed to mitigate the security challenges associated with cloud-based computations. TEE technologies provide isolated enclaves where sensitive computations can be performed securely, shielding them from potential threats and unauthorised access. Nonetheless, the adoption of these security technologies introduces a delicate balance between security, usability, and performance. On the one hand, the complexity of TEE technologies necessitates the creation of tools to enhance usability while upholding robust security guarantees. Conversely, the security architecture implemented by TEE technologies imposes computational overhead, thus impacting system performance.

In this dissertation, we aimed to achieve a compromise between security, usability, and performance in TEE-based applications. We first developed automatic code partitioning tools to reduce the trusted computing base (TCB) of enclave programs, thus reducing the potential attack surface that attackers could compromise to breach the security of the system. This not only tackles the security aspect, but further facilitates adoption for higher level languages, for which existing TEE technologies lack direct support. Subsequently, we shifted the focus to performance improvement, where we proposed ideas and tools to mitigate the overhead introduced by TEE technologies such as Intel SGX.

## THESIS SUMMARY

In Chapter 1, we began by contextualising the research work done, providing a clearer understanding of the rationale of this thesis. Subsequently, in Chapter 2 we presented key background concepts and technologies that served as the foundation for our work.

In Chapter 3, we introduced MONTSALVAT, a tool to automatically partition Java applications for enclaves. MONTSALVAT leverages code annotations to specify sensitive application classes, and performs bytecode transformations to partition the code. The transformed program is then AOT compiled to GraalVM native images that execute in and out of the enclave in a distributed fashion.

In Chapter 4, we proposed SECV, a more generic approach to analyse and partition applications for enclaves. SECV leverages GraalVM's Truffle framework to provide special AST nodes that can be used across a wide range of programming languages to specify sensitive information in a program. The resulting program is then dynamically analysed and partitioned into trusted and untrusted components that execute in and out of the enclave respectively. Our prototype implementation illustrates that the TCB can be reduced while improving the performance of the application.

In Chapter 5, we presented FORTRESS, a system to secure peripheral I/O in IoT systems. FORTRESS provides a generic blueprint to partition peripheral I/O drivers, illustrating the design with secure $I^2S$

sound processing.

We then moved the focus to performance enhancement in Chapter 6, where we proposed techniques and tools to leverage persistent memory (PM) in enclaves, to eschew costly enclave transitions for I/O operations. To improve fault-tolerance guarantees, we proposed a mirroring mechanism which involves creating encrypted mirror copies of enclave data structures in PM, and synchronising these copies with their enclave counterparts throughout the application's lifetime. We demonstrated the feasibility of this approach with secure machine learning model training in enclaves.

Lastly, in Chapter 7, we identified drawbacks with Intel SGX's switchless static configuration policy. We proposed ZC-SWITCHLESS, a dynamic system to drive SGX switchless calls while minimising waste of CPU resources. Extensive evaluations show ZC-SWITCHLESS obviates the performance penalty from misconfigured Intel SGX switchless systems, while minimising CPU waste.

## OUTLOOK

While this thesis addressed some security and performance challenges in TEEs, there are still many open research issues to be tackled in this field. Moreover, the tools we have proposed can be further extended to improve their functionality and usability on a broader scale. In the following, we outline potential avenues for future research work that build upon the foundations laid out in this thesis.

Firstly, a generic code partitioning framework like SECV provides a solid bedrock for future research as it tackles the language aspect which is key to enabling widespread adoption of TEE technologies. More specifically, POLYTAINT can be extended to fully cover more TRUFFLE languages like TruffleRuby, FastR, *etc*. Regarding LLVM-based languages like C/C++, the Truffle framework provides an LLVM interpreter, Sulong [150]. This interpreter can also be extended to run programs in Go, by leveraging tools like GoLLVM [53]. Alternatively, Truffle's WebAssembly (Wasm) implementation can be leveraged to provide a common compilation target for supported languages: Rust, Go, Kotlin, *etc*. While the taint analysis approach provided by POLYTAINT is language-agnostic by virtue of the TRUFFLE framework, new TRUFFLE languages may have some language-specific constructs which need to be taken into account for analysis to be carried out effectively. As a result, adding support for these languages in SECV mainly involves incorporating instrumentation nodes to handle these language-specific semantic constructs and AST nodes in the new language. There is in-depth documentation online to facilitate these extensions. Further, we could envision a hybrid approach comprising of both static and dynamic analysis for POLYTAINT's taint-tracking implementation, thereby providing much better code coverage.

As regards a tool like FORTRESS, static analysis techniques using a framework like Frama-C [48] can be incorporated so as to streamline the code partitioning process. Machine learning classification algorithms can also be introduced to automate data obfuscation. This will involve utilising pre-trained deep learning models within the trusted application to identify sensitive data in data streams, upon which appropriate obfuscation techniques may be applied.

As a more generic point, we could envision exploring additional applications and domains where the tools and frameworks proposed, *e.g.*, SECV, PLINIUS, ZC-SWITCHLESS could be employed. Investigating diverse use cases and real-world scenarios will contribute to a more comprehensive understanding of the potential impact and effectiveness of the proposed tools and techniques. This could involve collaboration with industry to implement the proposed approaches in production settings, ensuring their applicability beyond the academic realm. Given the current popularity of machine learning techniques, a secure ML framework like PLINIUS holds the potential for widespread utilization. To make this more feasibile, GPUs and TPUs can be leveraged to enhance the performance of expensive enclave operations. This will require security extensions in these accelerators, a direction which is currently being explored by companies like Nvidia [127].

While the bulk of the research work presented thus far focused on TEE technologies for process isolation like Intel SGX and Arm TrustZone, the current trend in TEE-related development suggests a noticeable shift towards TEEs for virtual machine isolation, *i.e.*, AMD SEV, Intel TDX, and more recently Arm CCA. As a result, there is a prospect to expand the tools introduced for the latter TEE technologies. More precisely, tools for code partitioning like SecV could be extended to partition programs which are executed in separate virtual machine instances, so as to achieve privilege separation. The performance enhancement techniques outlined in Chapter 6 and Chapter 7 can equally be applied to the programs executed within these VM instances.

Lastly, there is an opportunity to adopt formal verification techniques to rigorously prove the security properties of the tools implemented. This approach would add an additional layer of assurance to the security measures introduced in our research.

# PUBLICATIONS

## 2021

1. Boris Teabe, Peterson Yuhala, Alain Tchana, Fabien Hermenier, Daniel Hagimont, and Gilles Muller.
**(No)Compromis: paging virtualization is not a fatality**
in Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE).
Virtual, USA: ACM, April 2021, pp. 43–56.
DOI: 10.1145/3453933.3454013.

2. Peterson Yuhala, Pascal Felber, Valerio Schiavoni, and Alain Tchana.
**Plinius: Secure and Persistent Machine Learning Model Training**
in Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).
Virtual, Taipei, Taiwan: IEEE, June 2021, pp. 52–62.
DOI: 10.1109/DSN48987.2021.00022.

3. Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi.
**Montsalvat: Intel SGX shielding for GraalVM native images**
in Proceedings of the 22nd International Middleware Conference.
Virtual, Québec City, Canada: ACM, December 2021, pp. 352–364.
DOI: 10.1145/3464298.3493406.

## 2023

4. Peterson Yuhala, Michael Paper, Timothée Zerbib, Pascal Felber, Valerio Schiavoni, and Alain Tchana.
**SGX Switchless Calls Made Configless**
in Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).
Porto, Portugal: IEEE, June 2023, pp. 229–238.
DOI: 10.1109/DSN58367.2023.00032.

5. Peterson Yuhala.
**Enhancing IoT Security and Privacy with Trusted Execution Environments and Machine Learning**
in Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) - Supplemental Volume.
Porto, Portugal: IEEE, June 2023, pp. 176–178.
DOI: 10.1109/DSN-S58398.2023.00047.

6. Peterson Yuhala, Pascal Felber, Hugo Guiroux, Jean-Pierre Lozi, Alain Tchana, Valerio Schiavoni, and Gaël Thomas.
   **SecV: Secure Code Partitioning via Multi-Language Secure Values**
   in Proceedings of the 24th International Middleware Conference.
   Bologna, Italy: ACM, December 2023, pp. 207–219.
   DOI: 10.1145/3590140.3629116.

7. Jämes Ménétrey, Aeneas Grüter, Peterson Yuhala, Julius Oeftiger, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni.
   **A Holistic Approach for Trustworthy Distributed Systems with WebAssembly and TEEs**
   in Proceedings of the 27th International Conference on Principles of Distributed Systems (OPODIS).
   Tokyo, Japan: IEEE, December 2023, pp. 23:1–23:23.
   DOI: 10.4230/LIPIcs.OPODIS.2023.23.

## 2024

8. Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni.
   **Fortress: Securing IoT Peripherals with Trusted Execution Environments**
   in Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing (SAC)
   Avila, Spain: ACM, April 2024.
   DOI: 10.1145/3605098.3635994.

# BIBLIOGRAPHY

[1] Martìn Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283 (see pp. 89, 100).

[2] Alpine Linux. https://alpinelinux.org/. Accessed on 06-05-2020 (see p. 35).

[3] Alon Altman and Moshe Tennenholtz. Ranking systems: the PageRank axioms. In: *Proceedings of the 6th ACM conference on Electronic Commerce (EC 05)*. Vancouver, BC, Canada, 2005, pp. 1–8 (see pp. 40, 46, 61).

[4] Noah Apthorpe, Dillon Reisman, and Nick Feamster. A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic. In: *arXiv preprint arXiv:1705.06805* (2017) (see p. 67).

[5] Arm. Armv8-A Architecture Registers. https://developer.arm.com/documentation/ddi0595/2021-12/AArch64-Registers/CNTVCT-EL0--Counter-timer-Virtual-Count-register. August 23, 2023 (see p. 75).

[6] ARM-Software. Trusted Board Boot. https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/design/trusted-board-boot.rst. Accessed on 31-08-2023 (see p. 72).

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA, USA: USENIX Association, Nov. 2016, pp. 689–703 (see pp. 2, 4, 24, 25, 38, 43, 64, 86, 102, 118).

[8] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. In: *Computer Networks* 54.15 (2010), pp. 2787–2805. DOI: https://doi.org/10.1016/j.comnet.2010.05.010 (see p. 67).

[9] AWS. AWS IoT. https://aws.amazon.com/com/iot/. September 26, 2023 (see p. 71).

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 267–283 (see p. 109).

[11] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In: *ACM Transactions on Computer Systems (TOCS)* (2015) (see pp. 2, 24, 43).

[12] Jonathan Bell and Gail Kaiser. Dynamic Taint Tracking for Java with Phosphor (Demo). In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, 2015, pp. 409–413. DOI: `10.1145/2771783.2784768` (see p. 64).

[13] Daniele Bonetta. GraalVM: metaprogramming inside a polyglot system (invited talk). In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection (SPLASH 18)*. Boston, MA, USA, Nov. 2018, pp. 3–4 (see pp. 4, 25).

[14] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In: *Proceedings of the 17th International Middleware Conference (Middleware 16)*. Middleware '16. Trento, Italy: Association for Computing Machinery, 2016. DOI: `10.1145/2988336.2988350` (see pp. 3, 11, 43).

[15] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In: *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, Aug. 2004 (see p. 3).

[16] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: *USENIX Security 2018* (see pp. 26, 48, 91).

[17] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In: *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM 04)*. SIAM. Lake Buena Vista, FL, USA, 2004, pp. 442–446 (see pp. 40, 61).

[18] Somnath Chakrabarti, Thomas Knauth, Dmitrii Kuvaiskii, Michael Steiner, and Mona Vij. Chapter 8 - Trusted execution environment with Intel SGX. In: *Responsible Genomic Data Sharing*. Ed. by Xiaoqian Jiang and Haixu Tang. Academic Press, 2020, pp. 161–190. DOI: `https://doi.org/10.1016/B978-0-12-816197-5.00008-5` (see p. 8).

[19] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In: ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 253–264. DOI: `10.1145/2451116.2451145` (see p. 108).

[20] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. OPERA: Open Remote Attestation for Intel's Secure Enclaves. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 19)*. London, United Kingdom, Nov. 2019, pp. 2317–2331 (see p. 25).

[21] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID

voltage scaling interface. In: *30th USENIX Security Symposium (USENIX Security 21)*. Online, 2021 (see p. 25).

[22] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX Demystified: A Top-Down Approach. In: *arXiv preprint arXiv:2303.15540* (2023) (see pp. 2, 8).

[23] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. DriverGuard: Virtualization-Based Fine-Grained Protection on I/O Flows. In: *ACM Trans. Inf. Syst. Secur.* 16.2 (Sept. 2013). DOI: 10.1145/2505123 (see p. 79).

[24] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA '07. London, United Kingdom: Association for Computing Machinery, 2007, pp. 196–206. DOI: 10.1145/1273463.1273490 (see pp. 51, 64).

[25] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. ACES: Automatic Compartments for Embedded Systems. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 65–82 (see pp. 74, 79).

[26] IBM Cloud. Provisioning a bare metal server with Intel Software Guard Extension architecture. https://cloud.ibm.com/docs/bare-metal?topic=bare-metal-bm-server-provision-sgx. Accessed on 29-10-2023 (see p. 9).

[27] CNBC. Google admits partners leaked more than 1,000 private conversations with Google Assistant. https://www.cnbc.com/2019/07/11/google-admits-leaked-private-voice-conversations.html. April 3, 2023. 2019 (see p. 67).

[28] W.T. Cochran, J.W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, G.C. Maling, D.E. Nelson, C.M. Rader, and P.D. Welch. What is the fast Fourier transform? In: *Proceedings of the IEEE* 55.10 (1967), pp. 1664–1674. DOI: 10.1109/PROC.1967.5957 (see p. 38).

[29] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux device drivers. " O'Reilly Media, Inc.", 2005 (see p. 69).

[30] Intel Corporation. Intel SGX Developer Ref. for Linux. https://download.01.org/intel-sgx/sgx-linux/2.15/docs/Intel_SGX_Developer_Reference_Linux_2.15_Open_Source.pdf. Accessed on 23-10-2022 (see pp. 3, 14, 24, 57).

[31] Intel Corporation. Intel Software Guard Extensions Developer Reference for Linux OS. https://download.01.org/intel-sgx/sgx-linux/2.13/docs/Intel_SGX_Developer_Reference_Linux_2.13_Open_Source.pdf. 2018 (see pp. 8, 92, 95, 102–105, 118).

[32] Intel Corporation. SDK for Intel Software Guard Extensions SSL. https://github.com/intel/intel-sgx-ssl (see p. 110).

[33] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In: *SPAA'18* (see pp. 86, 88, 89, 93).

[34] Victor Costan and Srinivas Devadas. Intel SGX Explained. In: *IACR Cryptology ePrint Archive* 2016.86 (2016), p. 86 (see pp. 2, 4, 8–10, 12, 13, 25, 102).

[35] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 857–874 (see pp. 2, 3).

[36] Darknet: Open Source Neural Networks in C. `https://pjreddie.com/darknet/`. Accessed on 07-05-2020 (see pp. 89, 91, 100).

[37] Advanced Micro Devices. AMD Memory Encryption. `https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf`. Accessed on 11-07-2023 (see pp. 2, 8).

[38] Docker, inc. Docker: Empowering App Development for Developers. `https://www.docker.com`. Accessed on 10-05-2021 (see p. 43).

[39] Morris J Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Tech. rep. 2007 (see p. 88).

[40] Morris J Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Tech. rep. 2007 (see p. 92).

[41] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. In: *Cryptology ePrint Archive* (2012) (see p. 8).

[42] Pratik Fegade and Christian Wimmer. Scalable pointer analysis of data structures using semantic models. In: *Proceedings of the 29th International Conference on Compiler Construction (CC 2020)*. San Diego, CA, USA, 2020, pp. 39–50 (see p. 24).

[43] Fortanix. Fortanix Enclave Development Platform. `https://edp.fortanix.com/`. Accessed on 21-10-2023 (see p. 3).

[44] Apache Software Foundation. Apache Hadoop. `https://hadoop.apache.org/`. Accessed on 11-05-2021 (see pp. 3, 24).

[45] Apache Software Foundation. Apache Spark – Unified Analytics Engine for Big Data. `https://spark.apache.org/`. Accessed on 11-05-2021 (see pp. 3, 24).

[46] Apache Software Foundation. Apache ZooKeeper. `https://zookeeper.apache.org/`. Accessed on 11-05-2021 (see p. 24).

[47] OpenSSL Software Foundation. User Guide for the OpenSSL FIPS Object Module v2.0. `https://www.openssl.org/docs/fips/UserGuide-2.0.pdf`. Accessed on 10-01-2020 (see p. 110).

[48] FRAMA-C. A platform to make your C code safer and more secure. `https://frama-c.com/`. Accessed on 27-12-2023. 2023 (see p. 120).

[49] Adrien Ghosn, James R Larus, and Edouard Bugnion. Secured routines: Language-based construction of trusted execution environments. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 571–586 (see p. 46).

[50] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: *ICML'16* (see p. 100).

[51] GlobalPlatform. Specification Library. https://globalplatform.org/specs-library/. Accessed on 18-09-2023 (see p. 69).

[52] GNU. Built-in Functions for Memory Model Aware Atomic Operations. https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gcc/_005f_005fatomic-Builtins.html. 2021 (see p. 108).

[53] GoLLVM. https://go.googlesource.com/gollvm/. Accessed on 19-11-2023 (see p. 120).

[54] Google. Asylo. https://asylo.dev/. October 26, 2023 (see pp. 3, 24).

[55] Google. More information about our processes to safeguard speech data. https://www.blog.google/products/assistant/more-information-about-our-processes-safeguard-speech-data/. April 3, 2023. 2019 (see p. 67).

[56] Franz Gregor, Robert Krahn, Do Le Quoc, and Christof Fetzer. SinClave: Hardware-Assisted Singletons for TEEs. In: *Proceedings of the 24th International Middleware Conference*. Middleware '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 85–97. DOI: 10.1145/3590140.3629107 (see p. 12).

[57] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In: *SIGPLAN Not.* 51.2 (Oct. 2015), pp. 78–90. DOI: 10.1145/2936313.2816714 (see pp. 46, 47).

[58] William Grosso. Java RMI. O'Reilly Media, 2002 (see p. 24).

[59] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and Secure DNN Inference with Enclaves. In: *arXiv preprint arXiv:1810.00602* (2018) (see pp. 90, 95, 99).

[60] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In: *USENIX Security 2017* (see pp. 26, 48, 91).

[61] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. A Hardware-Software Co-design for Efficient Intra-Enclave Isolation. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3129–3145 (see pp. 2, 9).

[62] Shay Gueron. Memory Encryption for General-Purpose Processors. In: *IEEE Secur. Priv. 2016* (2016) (see p. 10).

[63] Saransh Gupta, Rosario Cammarota, and Tajana Šimunić Rosing. MemFHE: End-to-End Computing with Fully Homomorphic Encryption in Memory. In: *ACM Trans. Embed. Comput. Syst.* (Nov. 2022). DOI: 10.1145/3569955 (see p. 8).

[64] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold-Boot Attacks on Encryption Keys. In: *Commun. ACM* 52.5 (May 2009), pp. 91–98. DOI: 10.1145/1506409.1506429 (see p. 25).

[65] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. Mlcapsule: Guarded offline deployment of machine learning as a service. In: *arXiv preprint arXiv:1808.00590* (2018) (see p. 90).

[66] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Catherine Jones. Privacy-preserving machine learning in cloud. In: *Proceedings of the 2017 on Cloud Computing Security Workshop*. 2017 (see p. 100).

[67] Felicitas Hetzelt and Robert Buhren. Security Analysis of Encrypted Virtual Machines. In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 129–142. DOI: 10.1145/3050748.3050763 (see p. 2).

[68] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. In: *arXiv preprint arXiv:1803.05961* (2018) (see pp. 89, 90, 99).

[69] Nick Hynes, Raymond Cheng, and Dawn Song. Efficient deep learning on multi-source private data. In: *arXiv preprint arXiv:1807.06689* (2018) (see p. 100).

[70] Adam Ierymenko. Kissdb: simple stupid database. https://github.com/adamierymenko/kissdb. Accessed on 01-09-2022 (see p. 110).

[71] Intel. Intel Xeon Scalable Processors. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions-processors.html. Accessed on 28-11-2023. 2021 (see p. 97).

[72] Intel. Reference Architecture for Privacy Preserving Machine Learning with Intel SGX and TensorFlow Serving. https://www.intel.com/content/www/us/en/developer/articles/technical/privacy-preserving-ml-with-sgx-and-tensorflow.html. Accessed on 28-10-2022 (see p. 49).

[73] Intel Optane DC Persistent Memory. https://intel.ly/2zlnBTA. Accessed on 08-08-2019 (see p. 19).

[74] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements

of the intel optane DC persistent memory module. In: *arXiv preprint arXiv:1903.05714* (2019) (see pp. 84, 85).

[75] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous Isolated Execution for Commodity GPUs. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 455–468. DOI: 10.1145/3297858.3304021 (see p. 99).

[76] Bargav Jayaraman and David Evans. Evaluating Differentially Private Machine Learning in Practice. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1895–1912 (see p. 90).

[77] Jboss-Javassist. Javassist. https://www.javassist.org/. Accessed on 19-04-2021 (see p. 28).

[78] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In: *ACM Multimedia 2014* (see pp. 89, 100).

[79] Jianyu Jiang, Xusheng Chen, TszOn Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. Uranus: Simple, efficient SGX programming and its applications. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS 2020)*. Taipei, Taiwan, 2020, pp. 826–840 (see pp. 24, 25, 44, 46, 65).

[80] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. Supporting intel sgx on multi-socket platforms. In: *Intel Corp* (2021) (see p. 11).

[81] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: Epid provisioning and attestation services. In: *White Paper* (2016) (see p. 13).

[82] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. In: *Foundations and Trends in Machine Learning* 14.1–2 (2021), pp. 1–210 (see p. 90).

[83] Karnati. Data-in-use protection on IBM Cloud using Intel SGX. https://www.ibm.com/blogs/bluemix/2018/05/data-use-protection-ibm-cloud-using-intel-sgx/. Accessed on 15-05-2020 (see p. 9).

[84] Jonatan Kazmierczak. High performance at low cost – choose the best JVM and the best Garbage Collector for your needs. https://bit.ly/3f6mLj0. Accessed on 14-05-2021. 2020 (see p. 42).

[85] Keystone Enclave. https://docs.keystone-enclave.org/en/latest/Keystone-Applications/SDK-Basics.html. Accessed on 24-10-2023. 2021 (see pp. 3, 24).

[86] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In: *NDSS*. 2018 (see p. 74).

[87] Steven L Kinney. Trusted platform module basics: using TPM in embedded systems. Elsevier, 2006 (see p. 8).

[88] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, pp. 132–148. DOI: 10.1145/3492321.3519582 (see p. 64).

[89] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. In: *arXiv preprint arXiv:1801.05863* (2018) (see p. 13).

[90] David Kohlbrenner, Shweta Shinde, Dayeol Lee, Krste Asanovic, and Dawn Song. Building Open Trusted Execution Environments. In: *IEEE Security and Privacy* 18.5 (Sept. 2020), pp. 47–56. DOI: 10.1109/MSEC.2020.2990649 (see p. 3).

[91] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ Client Compiler for Java 6. In: *ACM Trans. Archit. Code Optim.* 5.1 (May 2008). DOI: 10.1145/1369396.1370017 (see p. 16).

[92] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. Multi-Language Dynamic Taint Analysis in a Polyglot Virtual Machine. In: *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*. MPLR 2020. Virtual, UK: Association for Computing Machinery, 2020, pp. 15–29. DOI: 10.1145/3426182.3426184 (see pp. 17, 47, 64).

[93] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Boston, MA, USA, 2012, pp. 31–46 (see pp. 3, 24, 38, 61).

[94] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. DOI: 10.1145/3342195.3387532 (see pp. 8, 67).

[95] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving Remote Deep-learning Inference Using SGX. In: *MobiCom'19* (see p. 99).

[96] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 2018, pp. 1–13 (see p. 79).

[97] Dingding Li, Ronghua Lin, Lijie Tang, Hai Liu, and Yong Tang. SGXPool: Improving the performance of enclave creation in the cloud. In: *Transactions on Emerging Telecommunications Technologies* (2019), e3735 (see p. 118).

[98] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential Serverless Made Efficient with Plug-In Enclaves. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 306–318. DOI: 10.1109/ISCA52012.2021.00032 (see pp. 109, 117).

[99] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 465–484 (see p. 9).

[100] Zinan Li, Wenhao Li, Yubin Xia, and Binyu Zang. TEEp: Supporting Secure Parallel Processing in ARM TrustZone. In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. 2020, pp. 544–553. DOI: 10.1109/ICPADS51040.2020.00076 (see p. 79).

[101] LibTomCrypt. https://github.com/libtom/libtomcrypt. September 26, 2023 (see p. 73).

[102] Arm Limited. Exception levels. https://developer.arm.com/documentation/102412/0103/Privilege-and-Exception-levels/Exception-levels. Accessed on 20-09-2023 (see pp. 3, 16).

[103] Linaro Limited. Open Source Secure Software. https://www.trustedfirmware.org/. Accessed on 18-09-2023 (see p. 16).

[104] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA, USA: USENIX Association, July 2017, pp. 285–298 (see pp. 3, 25, 44, 46, 65).

[105] LinkedIn. PalDB: an embeddable write-once key-value store written in Java. https://github.com/linkedin/PalDB. Accessed on 01-05-2021 (see pp. 24, 38).

[106] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. Software abstractions for trusted sensors. In: *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 2012, pp. 365–378 (see p. 79).

[107] Naiwei Liu, Meng Yu, Wanyu Zang, and Ravi Sandhu. On the Cost-Effectiveness of TrustZone Defense on ARM Platform. In: *Information Security Applications: 21st International Conference, WISA 2020, Jeju Island, South Korea, August 26–28, 2020, Revised Selected Papers*. Jeju Island, Korea (Republic of): Springer-Verlag, 2020, pp. 203–214. DOI: 10.1007/978-3-030-65299-9_16 (see p. 70).

[108] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In: *Proceedings of the Eleventh*

*European Conference on Computer Systems*. EuroSys ’16. London, United Kingdom: Association for Computing Machinery, 2016. DOI: 10.1145/2901318.2901326 (see p. 106).

[109] Mohammad Mahhouk, Nico Weichbrodt, and Rüdiger Kapitza. SGXoMeter: Open and Modular Benchmarking for Intel SGX. In: *Proceedings of the 14th European Workshop on Systems Security*. EuroSec ’21. Online, United Kingdom: Association for Computing Machinery, 2021, pp. 55–61. DOI: 10.1145/3447852.3458722 (see p. 117).

[110] Giovanni Mazzeo, Sergei Arnautov, Christof Fetzer, and Luigi Romano. SGXTuner: Performance Enhancement of Intel SGX Applications via Stochastic Optimization. In: *IEEE Transactions on Dependable and Secure Computing* (2021), pp. 1–1. DOI: 10.1109/TDSC.2021.3064391 (see pp. 103, 117).

[111] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave. In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. HASP ’16. Seoul, Republic of Korea: Association for Computing Machinery, 2016. DOI: 10.1145/2948618.2954331 (see p. 11).

[112] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In: *HASP’13* () (see pp. 9, 10).

[113] Larry W McVoy, Carl Staelin, et al. lmbench: Portable Tools for Performance Analysis. In: *USENIX annual technical conference*. San Diego, CA, USA. 1996, pp. 279–294 (see pp. 110, 114).

[114] Marcela S Melara, Michael J Freedman, and Mic Bowman. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. In: *arXiv preprint arXiv:1907.13245* (2019) (see p. 43).

[115] Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. Attestation Mechanisms for Trusted Execution Environments Demystified. In: *DAIS*. Vol. 13272. Lecture Notes in Computer Science. Springer, 2022, pp. 95–113 (see p. 75).

[116] Jämes Ménétrey, Aeneas Grüter, Peterson Yuhala, Julius Oeftiger, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. A Holistic Approach for Trustworthy Distributed Systems with WebAssembly and TEEs. In: *CoRR* abs/2312.00702 (2023). DOI: 10.48550/ARXIV.2312.00702. arXiv: 2312.00702 (see p. 6).

[117] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An Embedded Trusted Runtime for WebAssembly. In: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 205–216. DOI: 10.1109/ICDE51399.2021.00025 (see p. 64).

[118] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone. In: *ICDCS*. IEEE, 2022, pp. 1177–1189 (see p. 75).

[119] Microsoft. Confidential Computing on Azure. https://learn.microsoft.com/en-us/azure/confidential-computing/overview-azure-products. Accessed on 29-10-2023 (see p. 9).

[120] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. DarkneTZ: towards model privacy at the edge using trusted execution environments. In: *MobiSys*. ACM, 2020, pp. 161–174 (see p. 78).

[121] A. Mogage, R. Pires, V. Craciun, E. Onica, and P. Felber. Supply Chain Malware Targets SGX: Take Care of what you Sign. In: *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2019, pp. 52–528. DOI: 10.1109/SRDS47363.2019.00016 (see p. 25).

[122] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 19–38. DOI: 10.1109/SP.2017.12 (see pp. 49, 62).

[123] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In: *IEEE S&P 2017* (see pp. 89, 92, 95).

[124] Dominic P Mulligan, Gustavo Petri, Nick Spinale, Gareth Stockwell, and Hugo JM Vincent. Confidential Computing - a brave new world. In: *2021 international symposium on secure and private execution environment design (SEED)*. IEEE. 2021, pp. 132–138 (see pp. 2, 8).

[125] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How Persistent is your Persistent Memory Application? In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1047–1064 (see p. 86).

[126] NVIDIA. Source code of the OP-TEE OS for NVIDIA Jetson Linux. https://github.com/NVIDIA/optee_os-nvidia/. Accessed on 18-09-2023. 2023 (see p. 75).

[127] Nvidia. NVIDIA Confidential Computing. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/. 2023 (see p. 121).

[128] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. Ed. by Haryadi S. Gunawi and Benjamin Reed. Boston, MA, USA: USENIX Association, 2018, pp. 227–240 (see pp. 26, 48, 91).

[129] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Efficient formal verification for the Linux kernel. In: *Software Engineering and Formal Methods: 17th International Conference, SEFM 2019, Oslo, Norway, September 18–20, 2019, Proceedings 17*. Springer. 2019, pp. 315–332 (see p. 3).

[130] OpenEnclave. GitHub - openenclave/openenclave: SDK for developing enclaves. https://github.com/openenclave/openenclave. Accessed on 21-10-2023 (see pp. 3, 24).

[131] Oracle. Execution Event Node. https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/ExecutionEventNode.html. Accessed on 11-10-2022 (see p. 48).

[132] Oracle. Fast R. https://github.com/oracle/fastr. Accessed on 11-10-2022 (see p. 46).

[133] Oracle. Finalizers. https://www.graalvm.org/reference-manual/native-image/Limitations/. Accessed on 09-09-2021 (see p. 33).

[134] Oracle. Framework for ahead-of-time compilation with Native Image. https://github.com/oracle/graal/tree/master/substratevm. 2023 (see pp. 18, 42, 63).

[135] Oracle. Graal Python. https://github.com/oracle/graalpython. Accessed on 11-10-2022 (see p. 47).

[136] Oracle. GraalJS. https://github.com/oracle/graaljs. Accessed on 11-10-2022 (see p. 46).

[137] Oracle. GraalVM Chrome Debugger. https://www.graalvm.org/22.2/tools/chrome-debugger/. Accessed on 11-10-2022 (see pp. 17, 47).

[138] Oracle. GraalVM Native Image. https://www.graalvm.org/reference-manual/native-image/. Accessed on 19-04-2021. 2017 (see pp. 24, 29, 37).

[139] Oracle. GraalVM Profiling Command Line Tools. https://www.graalvm.org/22.2/tools/profiling/. Accessed on 11-10-2022 (see p. 47).

[140] Oracle. GraalVM SDK Java API Reference – CCharPointer. https://www.graalvm.org/sdk/javadoc/index.html?org/graalvm/nativeimage/c/type/CCharPointer.html. Accessed on 29-04-2021 (see p. 30).

[141] Oracle. GraalVM SDK Java API Reference – CEntryPoint. https://www.graalvm.org/sdk/javadoc/org/graalvm/nativeimage/c/function/CEntryPoint.html. Accessed on 29-04-2021 (see pp. 25, 57).

[142] Oracle. Interface Frame. https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/frame/Frame.html. Accessed on 15-10-2022 (see p. 55).

[143] Oracle. JSFunctionCallNode. https://github.com/oracle/graaljs/blob/master/graal-js/src/com.oracle.truffle.js/src/com/oracle/truffle/js/nodes/function/JSFunctionCallNode.java. Accessed on 15-10-2022 (see p. 54).

[144] Oracle. JSTags. https://github.com/oracle/graaljs/blob/master/graal-js/src/com.oracle.truffle.js/src/com/oracle/truffle/js/nodes/instrumentation/JSTags.java. Accessed on 15-10-2022 (see p. 54).

[145] Oracle. NodeObjectDescriptor. https://github.com/oracle/graalpython/blob/master/graalpython/com.oracle.graal.python/src/com/oracle/graal/

python/nodes/instrumentation/NodeObjectDescriptor.java. Accessed on 15-10-2022 (see p. 53).

[146] Oracle. Object Streams. https://docs.oracle.com/javase/tutorial/essential/io/objectstreams.html. Accessed on 20-10-2022 (see p. 57).

[147] Oracle. Polyglot Programming. https://www.graalvm.org/22.0/reference-manual/polyglot-programming/. Accessed on 18-02-2022 (see p. 47).

[148] Oracle. PythonCallNode. https://github.com/oracle/graalpython/blob/master/graalpython/com.oracle.graal.python/src/com/oracle/graal/python/nodes/call/PythonCallNode.java. Accessed on 01-03- 2022 (see p. 54).

[149] Oracle. Standard Tags. https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/StandardTags.html. Accessed on 11-10-2022 (see pp. 53, 54).

[150] Oracle. Sulong. https://github.com/oracle/graal/tree/master/sulong. Accessed on 11-10-2022 (see pp. 47, 120).

[151] Oracle. The Java Language Specification. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf. Accessed on 09-09-2021 (see p. 33).

[152] Oracle. Tracing agent. https://docs.oracle.com/en/graalvm/enterprise/19/guide/reference/native-image/tracing-agent.html. Accessed on 05-05-2021 (see p. 18).

[153] Oracle. Truffle Language Implementation Framework. https://www.graalvm.org/22.0/graalvm-as-a-platform/language-implementation-framework/. Accessed on 18-02-2022 (see pp. 4, 46).

[154] Oracle. Truffle Language Implementation Framework. https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/instrumentation/InstrumentableNode.WrapperNode.html. Accessed on 03-10-2022 (see p. 48).

[155] Oracle. Truffle Node. https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/Node.html. Accessed on 15-10-2022 (see pp. 54, 56).

[156] Oracle. Truffle Ruby. https://github.com/oracle/truffleruby. Accessed on 15-10-2022 (see p. 46).

[157] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: ACM, 2017, pp. 238–253. DOI: 10.1145/3064176.3064219 (see pp. 4, 84, 102, 108, 118).

[158] Persistent Memory Development Kit. https://github.com/pmem/pmdk. Accessed on 20-02-2020. 2020 (see p. 86).

[159] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. In: *ACM Comput. Surv.* 51.6 (Jan. 2019). DOI: 10.1145/3291047 (see pp. 2, 8, 9, 15).

[160] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. In: *arXiv preprint arXiv:1908.11143* (2019) (see pp. 2, 24, 43, 64).

[161] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 264–278 (see p. 2).

[162] OpenSSL Project. OpenSSL: EVP-aes-256-cbc. https://www.openssl.org/docs/manmaster/man3/EVP_aes_256_cbc.html (see p. 113).

[163] The Gramine Project. Graphene-SGX library OS. https://github.com/gramineproject/graphene. 2023 (see pp. 42, 63).

[164] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. SecureTF: A Secure TensorFlow Framework. In: *Proceedings of the 21st International Middleware Conference (Middleware 20)*. Middleware '20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 44–59. DOI: 10.1145/3423211.3425687 (see pp. 2, 49, 100).

[165] Ronald Rivest and S Dusse. The MD5 message-digest algorithm. 1992 (see p. 29).

[166] Andy Rudoff. Persistent memory programming. In: *Login: The Usenix Magazine* 42.2 (2017), pp. 34–40 (see pp. 85, 86).

[167] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308 (see pp. 24, 52).

[168] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 175–188 (see p. 2).

[169] Steve Scargall. Programming persistent memory: A comprehensive guide for developers. Springer Nature, 2020 (see p. 18).

[170] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In: *2015 IEEE Symposium on Security and Privacy (SSP 15)*. IEEE. San Jose, CA, USA, 2015, pp. 38–54 (see pp. 3, 25, 43).

[171] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2017 (see pp. 26, 48, 91).

[172] Intel SGX SDK. https://tinyurl.com/95xcmak (see p. 109).

[173] HEX-Five Security. MultiZone Security for RISC-V. https://hex-five.com/multizone-security-tee-riscv/. Accessed on 06-11-2023 (see p. 2).

[174] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In: *NDSS'17*. 2017 (see p. 9).

[175] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 955–970. DOI: 10.1145/3373376.3378469 (see p. 2).

[176] Shweta Shinde. Securing Applications From Untrusted Operating Systems Using Enclaves. PhD thesis. National University of Singapore, 2018 (see p. 90).

[177] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. San Diego, CA, USA: The Internet Society, 2017 (see pp. 3, 24, 44).

[178] Yannis Smaragdakis and George Balatsouras. Pointer Analysis. In: *Found. Trends Program. Lang.* 2.1 (Apr. 2015), pp. 1–69. DOI: 10.1561/2500000014 (see p. 31).

[179] SPEC. SPECjvm2008. https://www.spec.org/jvm2008/. Accessed on 11-05-2021. 2021 (see p. 42).

[180] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. Extracting Taint Specifications for JavaScript Libraries. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 198–209. DOI: 10.1145/3377811.3380390 (see p. 64).

[181] Ron Stajnrod, Raz Ben Yehuda, and Nezer Jacob Zaidenberg. Attacking TrustZone on devices lacking memory protection. In: *J. Comput. Virol. Hacking Tech.* 18.3 (2022), pp. 259–269. DOI: 10.1007/s11416-021-00413-y (see p. 16).

[182] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 16)*. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 665–678. DOI: 10.1145/3173162.3177155 (see pp. 4, 11).

[183] Boris Teabe, Peterson Yuhala, Alain Tchana, Fabien Hermenier, Daniel Hagimont, and Gilles Muller. (No)Compromis: Paging Virtualization is Not a Fatality. In: *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 43–56. DOI: 10.1145/3453933.3454013 (see p. 6).

[184] Abhradeep Guha Thakurta. Differential Privacy: From Theory to Deployment. In: Vancouver, BC: USENIX Association, Aug. 2017 (see p. 90).

[185] The MNIST Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/. Accessed on 07-05-2020 (see p. 95).

[186] The TensorFlow Open Source Project on Open Hub. https://www.openhub.net/p/tensorflow. Accessed on 11-12-2020 (see p. 100).

[187] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless Calls Made Practical in Intel SGX. In: *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX 18)*. SysTEX '18. Toronto, Canada: ACM, 2018, pp. 22–27. DOI: 10.1145/3268935.3268942 (see pp. 4, 103, 118).

[188] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. SGXKernel: A Library Operating System Optimized for Intel SGX. In: *Proceedings of the Computing Frontiers Conference*. CF'17. Siena, Italy: Association for Computing Machinery, 2017, pp. 35–44. DOI: 10.1145/3075564.3075572 (see p. 118).

[189] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In: *ICLR'19* (see p. 99).

[190] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 601–618 (see p. 90).

[191] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 87–97. DOI: 10.1145/1542476.1542486 (see p. 64).

[192] TrustedFirmware.org. Pseudo trusted applications. https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html. Accessed on 02-04-2023 (see pp. 67, 69).

[193] TrustedFirmware.org. Secure Storage. https://optee.readthedocs.io/en/latest/architecture/secure_storage.html. Accessed on 20-09-2023. 2023 (see p. 74).

[194] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. Ed. by Dilma Da Silva and Bryan Ford. Santa Clara, CA, USA, 2017, pp. 645–658 (see p. 24).

[195] Chia-che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In: *USENIX ATC 2017* (see pp. 2, 3, 43, 44, 64, 86).

[196] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E Porter. Civet: An efficient java partitioning framework for hardware enclaves. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 505–522 (see p. 3).

[197] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In: *29th USENIX Security Symposium (USENIX Security 20)*. Online: USENIX Association, Aug. 2020, pp. 505–522 (see pp. 24, 25, 44, 46, 65).

[198] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. In: *Computer* 38.5 (2005), pp. 48–56 (see p. 3).

[199] Michael L Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. In: *Art Sci. Eng. Program.* 2.3 (2018), p. 14 (see pp. 16, 47).

[200] Peter VanNostrand, Ioannis Kyriazis, Michelle Cheng, Tian Guo, and Robert J. Walls. Confidential Deep Learning: Executing Proprietary Models on Untrusted Devices. In: *ArXiv* abs/1908.10730 (2019) (see p. 67).

[201] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully homomorphic encryption compilers. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1092–1108 (see pp. 8, 90).

[202] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In: *ASPLOS'11*. 2011 (see p. 86).

[203] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 681–696 (see p. 99).

[204] Intel VTune Profiler: Find and Fix Performance Bottlenecks Quickly and Realize All the Value of Your Hardware. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html` (see p. 118).

[205] Cheng Wang, Qianlin Liang, and Bhuvan Urgaonkar. An empirical analysis of amazon EC2 Spot Instance features affecting cost-effective resource procurement. In: *TOMPECS'18* (2018) (see pp. 90, 98).

[206] Yongzhi Wang, Yulong Shen, Cuicui Su, Ke Cheng, Yibo Yang, Anter Faree, and Yao Liu. CFHider: Control Flow Obfuscation with Intel SGX. In: *IEEE Conference on Computer Communications (INFOCOM 2019)*. Paris, France, 2019, pp. 541–549. DOI: `10.1109/INFOCOM.2019.8737444` (see p. 44).

[207] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanovic. The RISC-V instruction set manual volume II: Privileged architecture version 1.7. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49* (2015) (see p. 3).

[208] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In: *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*. Ed. by Paulo Ferreira and Liuba

Shrira. Rennes, France: ACM, 2018, pp. 201–213. DOI: 10.1145/3274808.3274824 (see p. 117).

[209] Samuel Weiser and Mario Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. CODASPY '17. Scottsdale, Arizona, USA: Association for Computing Machinery, 2017, pp. 261–268. DOI: 10.1145/3029806.3029822 (see p. 79).

[210] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 81–93. DOI: 10.1145/3140659.3080208 (see pp. 4, 38, 84, 102, 107, 108, 118).

[211] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. In: *2019 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 19)*. Athens, Greece, 2019 (see pp. 17, 24, 31, 58).

[212] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In: Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 187–204. DOI: 10.1145/2509578.2509581 (see pp. 4, 46).

[213] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In: *SIGPLAN Not.* 48.2 (Oct. 2012), pp. 73–82. DOI: 10.1145/2480360.2384587 (see p. 47).

[214] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. Intel Software Guard Extensions (Intel SGX) Software Support for Dynamic Memory Allocation inside an Enclave. In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. HASP 2016. Seoul, Republic of Korea: Association for Computing Machinery, 2016. DOI: 10.1145/2948618.2954330 (see p. 11).

[215] Jian Xu and Steven Swanson. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. In: *Proceedings of the 14th Usenix Conference on File and Storage Technologies*. FAST'16. Santa Clara, CA: USENIX Association, 2016, pp. 323–338 (see p. 85).

[216] Wenju Xu, Baocang Wang, Jiasen Liu, Yange Chen, Pu Duan, and Zhiyong Hong. Toward Practical Privacy-Preserving Linear Regression. In: *Inf. Sci.* 596.C (June 2022), pp. 119–136. DOI: 10.1016/j.ins.2022.03.023 (see pp. 46, 61).

[217] Zhenyu Xu, Thomas Mauldin, Qing Yang, and Tao Wei. Runtime Detection of Probing/Tampering on Interconnecting Buses. In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2021, pp. 247–251. DOI: 10.1109/FCCM51124.2021.00038 (see p. 25).

[218] Jinfeng Yang, Bingzhe Li, and David J. Lilja. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 5.1 (Feb. 2020). DOI: 10.1145/3372783 (see p. 19).

[219] Sangho Yi, Derrick Kondo, and Artur Andrzejak. Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud. In: *IEEE Cloud Computing 2010* (see p. 98).

[220] P. Yuhala, P. Felber, V. Schiavoni, and A. Tchana. Plinius: Secure and Persistent Machine Learning Model Training. In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2021, pp. 52–62. DOI: 10.1109/DSN48987.2021.00022 (see p. 44).

[221] P. Yuhala, M. Paper, T. Zerbib, P. Felber, V. Schiavoni, and A. Tchana. SGX Switchless Calls Made Configless. In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2023, pp. 229–238. DOI: 10.1109/DSN58367.2023.00032 (see pp. xiii, 6).

[222] Peterson Yuhala, Pascal Felber, Hugo Guiroux, Jean-Pierre Lozi, Alain Tchana, Valerio Schiavoni, and Gaël Thomas. SecV: Secure Code Partitioning via Multi-Language Secure Values. In: *Proceedings of the 24th International Middleware Conference on ZZZ*. Middleware '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 207–219. DOI: 10.1145/3590140.3629116 (see pp. xiii, 3, 5).

[223] Peterson Yuhala, Pascal Felber, Valerio Schiavoni, and Alain Tchana. Plinius: Secure and Persistent Machine Learning Model Training. In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2021, pp. 52–62. DOI: 10.1109/DSN48987.2021.00022 (see pp. xiii, 5, 49, 62).

[224] Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. Fortress: Securing IoT Peripherals with Trusted Execution Environments. In: *CoRR* abs/2312.02542 (2023). DOI: 10.48550/ARXIV.2312.02542. arXiv: 2312.02542 (see pp. xiii, 5).

[225] Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi. Montsalvat: Intel SGX Shielding for GraalVM Native Images. In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 352–364. DOI: 10.1145/3464298.3493406 (see pp. xiii, 5, 57, 61, 65).

[226] Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi. Montsalvat: Intel SGX Shielding for GraalVM Native Images. In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 352–364. DOI: 10.1145/3464298.3493406 (see p. 3).

[227] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 897–912 (see pp. 19, 86).

[228] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA, USA, 2017, pp. 283–298 (see pp. 2, 44).

[229] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, 2023, pp. 247–267 (see p. 2).

## COLOPHON

This document was typeset in LaTeX using the typographical look-and-feel `classicthesis` by André Miede.[1] The plots in this thesis were generated using `Gnuplot` and the figures were made with `Lucidspark`. The bibliography is typeset using `BibLaTeX`.

1 https://miede.de/