**Title**
Secure Computing using Certified Software and Trusted Hardware



**Author**
Sinha, Rohit

**Publication Date**
2017

Peer reviewed|Thesis/dissertation

**Secure Computing using Certified Software and Trusted Hardware**

By

Rohit Sinha

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor David Wagner
Professor George Necula
Associate Professor Antonio Montalban

Fall 2017

**Secure Computing using Certified Software and Trusted Hardware**

Abstract

**Secure Computing using Certified Software and Trusted Hardware**

by

Rohit Sinha

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Building applications that ensure confidentiality of sensitive data is a non-trivial task. Such applications constantly face threats due to vulnerabilities in the application's code, or infrastructure attacks due to malicious datacenter insiders and exploits in the lower computing layers (i.e. OS, hypervisor, BIOS, firmware) that the application relies upon.

This dissertation presents a novel approach for developing and verifying applications with provable confidentiality guarantees, even in the presence of such privileged adversaries. Our primary defense against infrastructure attacks is the use of trusted primitives such as Intel SGX *enclaves*, for isolating sensitive code and data within protected memory regions; enclaves are inaccessible to all other software running on the machine (i.e. OS, hypervisor, etc.), thus removing these large software layers from the trusted computing base (TCB). A central question addressed by this thesis is how the trusted hardware primitives can be used safely to build the trusted components of modern applications with provable guarantees. Prior experience suggests that even expert developers write unsafe programs that leak sensitive data due to programming errors and side channel attacks. To address this problem, this thesis makes contributions in formal threat models, modeling and specification of trusted platforms, and techniques to verify confidentiality properties of enclave programs.

First, this thesis formalizes adversary models, an abstract, interface-level model of trusted platforms (including Intel SGX and MIT Sanctum), and formal semantics of enclave execution. This formal framework is required for reasoning about a program's behavior in the presence of a privileged adversary. Next, this thesis presents tools and techniques for certifying confidentiality (at the binary level), a property that we decompose into the following desiderata: 1) lack of explicit leak of secrets via enclave's outputs, 2) protection against certain side-channel leaks — we only remove leaks via page-level memory access pattern, which is a new channel for privileged adversaries. For both desiderata, we develop verification tools and evaluate them on application's binaries including Map-Reduce programs from the Microsoft VC3 system, SPEC benchmarks, and several machine learning algorithms.

*Dedicated to Mummy and Papa*

# Contents

# List of Figures

# List of Tables

# Acknowledgments

First, and foremost, I thank Sanjit Seshia, my advisor, for his unwavering support and brilliant guidance during my entire Ph.D. While I am grateful to him for introducing me to formal verification, I am most thankful for teaching me how to do conduct and present research, and the importance of asking the right questions. I am truly fortunate to have been his student, and he will continue to be one of my role models in future. Thank you, Sanjit!

I would also like to thank Professor Antonio Montalban, Professor George Necula, and Professor David Wagner for providing valuable feedback on the dissertation and my qualifying exam. I am really grateful for the insightful advice David has given me during our collaboration in the early years of my Ph.D.

I thank Dr. Sriram Rajamani for all the discussions we had, which have shaped the key ideas presented in this dissertation — I will especially remember the excellent questions he would ask, which would often force me to clarify my own thought process. He has provided brilliant feedback on various components of this dissertation, and I hope to continue our work together.

I am also thankful to my mentors Dr. Manuel Costa, Dr. Akash Lal, Dr. Nuno Lopes, and Dr. Kapil Vaswani, all of whom mentored me during my internship at Microsoft Research (Cambridge, UK) in the summer of 2015, and for many months following that. The discussions we had helped shape the technical ideas behind the verification tools presented in this thesis. Akash shared his immense wealth of experience in tailoring various verification techniques to the challenging problems that we tackled, and his perseverance in making our method more scalable set an example that I hope to emulate. Manuel taught me the importance of system design in building provably correct systems. Nuno's brilliant suggestions for efficient SMT-based verification, and Kapil's deep knowledge of systems and programming languages were extremely valuable to the ideas that we developed together.

In the later stages of this dissertation, I also had the privilege of working with Professor Srini Devadas; his unbridled enthusiasm and knowledge of computer security was truly fascinating. I am also fortunate to work closely with Dr. Pramod Subramanyan and Ilia Lebedev on the formal verification of trusted platforms, which formed a core component of this dissertation.

Furthermore, while our work together was not included in this thesis, I am grateful to have worked with Nicholas Carlini, Thurston Dang, Sakshi Jain, Dr. Petros Maniatis, Michael

McCoyd, Professor Cynthia Sturton, Wei Yang Tan, and Prof. David Wagner. During my other internships, I also had the opportunity to work closely with Dr. Shuvendu Lahiri, Dr. Chris Hawblitzel, and Dr. Rebekah Leslie-Hurd. The long list of things I have learned from them have helped me along the way till date.

Over the course of my Ph.D., Berkeley has given me the joy of working alongside brilliant students of the learn-verify group: Daniel Holcomb, Susmit Jha, Wenchao Li, Alexander Donze, Daniel Bundala, Markus Rabe, Rudiger Ehlers, Tomosso Dreossi, Indranil Saha, Yichin Wu, Garvit Juniwal, Nishant Totla, Zach Wasson, Dorsa Sadigh, Ankush Desai, Marcell Chanlatte, Ben Caulfeld, Daniel Fremont, Katia Patkin, Eric Kim, Wei Yang Tan, and Jonathan Kotker.

Finally, I am grateful to my entire family, for their unconditional love and support throughout this endeavor.

This research was supported in part by SRC contract 2460.001, SRC contract 2638.001, NSF STARSS grant 1528108, and by Microsoft Research.

# Chapter 1

# Introduction

Modern software services are increasingly reliant on the public cloud for economic benefits including high availability, low maintenance, and on-demand scaling. Despite these advantages of cloud-based computing, there is a unanimous agreement amongst its users about the security and privacy concerns, especially when processing sensitive data. Over the years, users of cloud computing have suffered from various attacks and data breaches, exposing millions of sensitive customer records [66, 35].

One primary cause of these attacks is that a typical cloud-based service contains large software layers in its trusted computing base (TCB) — the TCB includes privileged computing layers such as the OS and Hypervisor, and parts of the application stack (e.g. web server), and each such layer can amass several millions of lines of code in standard deployments [2]. In the last few years, numerous exploits have been performed on these vulnerable software layers, allowing privileged malware to execute [33, 63]. Moreover, a malicious datacenter insider can exploit administrative privileges to log into machines with root access, and inspect sensitive contents of disks and memory. In short, by controlling the privileged software layers, the attacker can observe and tamper the contents of an application's memory, extract sensitive data, and cause security-critical applications to misbehave — we refer to an attacker that has compromised the privileged computing layers as a *privileged software adversary*. In addition to these platform-level attacks, modern applications are increasingly complex, use unsafe programming methods, and contain large volumes of code with unspecified behavior. Not surprisingly, attackers repeatedly find such applications to have critical vulnerabilities such as insufficient access control [131], unsafe cryptographic protections [29], and memory corruption bugs [117].

Recognizing these problems, processor vendors are now shipping CPUs with hardware primitives, such as Intel SGX *enclaves* [61], for isolating sensitive code and data within protected memory regions which are inaccessible to all other software running on the machine — we refer to the platform that implements these trusted primitives as an *enclave platform*. To support isolated execution, the enclave platform monitors all accesses to the enclave: only code running in the enclave can access the enclave's memory. As an example, VC3 [107] runs Map-Reduce analytics on an untrusted cloud by computing map and reduce functions on

sensitive cleartext data within enclaves, while the rest of the Hadoop stack (comprising over million lines of code) is untrusted, manages only encrypted data, and is developed using legacy software toolchains. In addition to enabling isolated execution, the enclave platform implements primitives for generating attested statements: code inside an enclave can get messages signed using a per-processor private key along with a hash-based measurement of the enclave. This allows other trusted entities to verify that messages originated from the desired (untampered) enclave running on a genuine platform. Finally, we require the enclave platform to implement a cryptographically secure entropy source which the enclave can use for cryptographic operations such as key generation. From here on, we use the term enclave to mean a program that enjoys the aforementioned properties of isolated execution, attested statements, and entropy source. At the time of writing, enclaves are implemented by several mainstream platforms such as Intel SGX [61] and Microsoft Hyper-V Virtual Secure Mode [134], and academic platforms such as MIT's RISC-V based Sanctum [38].

In this new paradigm, enclaves are the only trusted components of an application, providing us the luxury of programming them with greater rigor and stronger defenses. However, it is open research question as to how cloud services can be built using using hardware-based enclaves, while ensuring a tiny TCB and formal security guarantees even in the presence of a privileged software attacker. While enclave platforms implement the necessary primitives for trustworthy computing on sensitive data, these primitives are not sufficient by themselves. The enclave program must be implemented carefully, as vulnerabilities in the enclave program can be exploited to extract sensitive data or perform malicious actions, thus defeating the purpose of using enclave platforms in the first place. The enclave program must be secure, which includes, at the very least, a provable guarantee that sensitive data is never leaked to the attacker in any execution, under any potential interaction with the attacker — we refer to this property of the enclaves as confidentiality. However, ensuring confidentiality of enclave programs (or software in general) is non-trivial. In order to be useful, an enclave program interacts with the external world (via the untrusted host platform) using communication and storage mechanisms such as the network, file system, and shared memory; such interactions can reveal a program's secrets via explicit and implicit channels. An adversarial OS can tamper inputs, observe outputs, control resource allocation and scheduling, and these capabilities give an attacker a mechanism to interact with the enclave and potentially trigger vulnerabilities in its code [33]. Furthermore, due to hardware limitations, an attacker may observe several implicit attributes of an enclave program's execution, called side channel observations. For instance, on an Intel SGX platform, an attacker can use a compromised OS to observe cache hits and misses, page-level memory access patterns, execution time, and so on [76, 125] — these observations may reveal secrets for programs that compute on sensitive data. The burden of programming enclaves correctly and ensuring confidentiality, for both outputs and side channel observations, remains with the programmer.

We believe that the one of main challenges of developing secure enclaves is the lack of development and verification tools for building enclaves that are secure by construction. To reason about confidentiality properties, an enclave developer must a have precise understanding of the formal specification of the primitives offered by the enclave platform, which

requires the developer to reason about the low-level contract between the hardware and software layers. At the time of writing, there has only been informal security analysis of enclave programs and platforms, and informal characterization of the threat model. Moreover, the developer needs to guard against all potential operations (e.g. tampering of inputs and persistent storage [33]) and observations (including side channels) of the privileged software attacker. At the time of writing, there are no proposed techniques or tools for verifying security properties of enclave programs, other than those described in this thesis.

This dissertation makes several novel contributions towards addressing these challenges. We present techniques for developing enclave programs and verifying their confidentiality properties, at the level of machine code execution to maintain a tiny TCB, all-the-while defending against the operations and observations of a privileged software adversary. Part I of this dissertation develops a formal framework to reason about an enclave's execution in the presence of a privileged software adversary. Specifically, we present a formal semantics of enclave programs, a formal model of the enclave platform and the specification of its primitives, and a formal model of privileged software adversary. Part II of this dissertation presents a novel methodology and practical tools (including compilers and static analyzers) for safely programming enclaves and formally certifying the absence of vulnerabilities that leak secrets, both via explicit outputs and certain side channels.

## 1.1 Thesis Statement

*Through formal modeling of trusted enclave platforms, and compilation and verification tools for developing secure enclave programs, we can build software services that can be certified, at the level of machine code execution, to not leak sensitive data to a privileged software adversary.*

## 1.2 Thesis Contributions

Although enclave platforms provide the necessary primitives for developing outsourced software services, there is little research on methods for writing safe enclave programs that provide end-to-end security guarantees with a tiny TCB. We wish for these software services to provide formal guarantees of security (e.g. confidentiality) at the machine code level of execution. At a high level, we approach this goal by formally specifying the hardware-software interface, and providing tools to develop secure software layers on top. This degree of assurance requires, at the very least, a formal specification of the trusted platform primitives, the semantics of enclave program execution, and a formal model of the privileged software attacker, all of which are used to reason about the safety of enclave software — this is the core contribution of Part I of this thesis. Confidentiality of enclave programs requires protecting secrets, which needs understanding of the contract between the enclave developer and the enclave platform, and a methodology for writing enclave programs that ensures that secrets are not leaked, either explicitly via outputs or via side channels. Developing enclaves

that guarantee confidentiality of sensitive data against privileged attackers is challenging, because of the large set of powerful adversarial operations and observations, thus making it non-trivial to statically analyze enclave programs or defend them. To account for the privileged attacker's capabilities, developers need compiler and static analysis tools to automatically prevent or detect vulnerabilities in their code. The core contribution of part II of this thesis is a set of techniques and automatic tools to detect vulnerabilities that violate confidentiality, and also generate enclave programs that are correct by construction. We describe the contributions of part I and part II in further detail below.

### 1.2.1 Modeling and Verification of Enclave Programs, Platforms, and Adversaries

Despite growing interest, there has only been informal security analysis of enclave programs, platforms, and threat models. This lack of formalization has several consequences. Developers of enclave programs cannot formally reason about security of their programs: incorrect use of hardware primitives or informal characterization of attacker's abilities can lead to vulnerabilities in the enclave program. Furthermore, hardware designers cannot formally state security properties of their architectures and are unable to reason about potential vulnerabilities of enclaves running on their hardware. This part of the thesis bridges these gaps by developing a formal framework to reason about an enclave's execution in the presence of a privileged software adversary.

Specifically, we present a formal semantics of enclave program execution (Chapter 3), and a formal model of the enclave platform and the privileged software adversaries (Chapter 4). Furthermore, to generalize our guarantees beyond a specific platform, we propose an abstract enclave platform (Chapter 4) whose operations simulate the set of primitives offered by mainstream enclave platforms, such as Intel SGX and Sanctum. We then use these formal models to establish a theorem (Chapter 5) that any enclave program enjoys a set of security properties when run on the abstract platform (and consequently, SGX, Sanctum, etc.), thus enabling us to reason about an enclave's execution on a wide variety of enclave platforms. The contributions in this part of the thesis are vital to our goal of certifying confidentiality properties of enclave software with a tiny TCB.

### 1.2.2 Design and Verification Techniques for Enclave Programs

Practical defenses against a privileged adversary is an open research problem, which we address in this part of the thesis. Enclaves rely on the compromised host OS for I/O interactions with remote parties, scheduling, and resource management, and such interactions can reveal secrets, either directly or via side channels. The burden of programming enclaves correctly and ensuring confidentiality remains entirely with the programmer. First, the enclave developer must follow the enclave creation guidelines so that the hardware protects the enclave from an attacker that has gained privileged access to the system. Even then, the enclave developers needs to ensure that their code does not leak secrets via any vulnerability.

For instance, they should encrypt secrets before writing them to non-enclave memory. They should account for adversary modifying non-enclave memory at any time, which could result in time-of-check-to-time-of-use attacks [126]. They should program defensively to avoid any memory safety errors (e.g. buffer overflow, control flow hijacking, etc.). Writing safe enclaves that avoid such attacks is non-trivial, and the history of computer security indicates that even expert developers make critical errors such as memory corruption bugs [117], unsafe use of cryptographic operations, insufficient access control, information flow leaks, etc. Chapter 7 and Chapter 8 present programming methodologies and automatic tools (compilers and machine code verifiers) to both detect and prevent such vulnerabilities within enclave programs.

To add to the developer's woes, some mainstream enclave platforms do not provide idealized execution i.e. while they may offer isolated execution (with memory integrity), they do not necessarily offer strict confidentiality. For instance, Intel SGX leaks information about the enclave's execution via various software and hardware side channels (e.g. cache timing attacks, page-level memory access patterns), and mitigating these leaks requires even greater understanding of the enclave platform, careful programming practices, and automatic tool support. Closing the side channel of page-level access patterns is the focus of Chapter 9, which presents a toolchain consisting of a compiler and verifier to automatically produce machine code that eliminates such side channel leaks.

In summary, this part of the thesis presents a novel methodology for safely programming enclaves and formally certifying the absence of vulnerabilities that leak secrets, both via explicit outputs and certain side channels. As per our thesis statement, we wish to formally certify confidentiality of the enclave, and we verify this property at the machine code level to achieve a tiny software TCB — with verified machine code, our software TCB only includes the verification tools themselves, which are built upon decision procedures for First Order Logic theories.

## 1.3 Related Work

This dissertation presents a novel methodology for building provably secure services with a tiny software TCB. The related work presented in this section positions this dissertation in relation to alternate approaches for trustworthy computing and software security. Note that there are large bodies of related work that relate to the specific contributions and technical matter unveiled in the later chapters, and we discuss them within those chapters.

**Trustworthy Computing** Our overarching goal is to outsource computation on sensitive data, without the attacker tampering with the computation or learning the sensitive data. There are large bodies of research in the areas of cryptography and secure systems that actively pursue this goal, albeit from different directions and assumptions.

Fully homomorphic encryption (FHE) [97], usually referred to as the holy grail of cryptography, is a scheme for outsourcing computation directly on encrypted data. FHE allows

an untrusted server to compute arbitrary, efficiently computable functions over encrypted data, without access to the decryption key. The server produces an encrypted result that can be decrypted by the client, such that the decrypted result matches the result of that computation if it were performed directly on the plaintext input. However, the proposed schemes [53] are several orders of magnitude slower than the plaintext computation, which is a prohibitive overhead for practical applications — while some partial homomorphic schemes are practical, they only support limited operations (e.g. addition in the Paillier cryptosystem) which are well short of the functionality required for modern applications. Furthermore, adding integrity guarantees to FHE leads to a further slowdown of several orders of magnitude. Similarly, verifiable computing [52] allows a client to outsource a function to the untrusted server, which produces a proof of correct evaluation to be verified b the client. Similar to FHE, schemes for verifiable computing have significant performance overheads, thus limiting their use in practical systems.

On the other end of the spectrum, there have been groundbreaking results on building verified, trustworthy systems from the ground up. To defend against attacks on the privileged software infrastructure, this line of work explores the use of formal methods to build provably secure operating systems. The Ironclad project [56] and the Certikos project [55] both built a fully verified software stack (including OS, device drivers, and cryptographic libraries) from the ground-up. The seL4 project [65] built a microkernel and formally verified (at the machine code level) that it provides isolation between different applications. However, verifying system-wide properties at such low levels of abstraction requires significant amount of manual effort and creative insight; the seL4 verification consumed nearly 22 human years of effort [65]. Furthermore, because of this bottleneck, many of these efforts fall short of the functionality and performance offered by contemporary commercial solutions. We contend that this approach — i.e., building verified privileged software layers, which provide strong guarantees but limited functionality and performance — is unlikely to scale to real-world OS and system software.

We take a middle ground approach by taking advantage of recent developments in secure processors. Most recently, Intel SGX [79, 8, 58] implemented primitives for creating protected memory regions, called enclaves, that contains code and data which is inaccessible to any code outside of the enclave (including privileged software layers such as the OS). In addition to SGX, there have been several other commercial deployments of secure processors. ARM TrustZone [7] implements secure mode of execution to effectively create a single privileged enclave in an isolated address space. Academic work seeking to improve the security of aspects of conventional processors is also abundant [38, 75, 74, 46, 90]. These platforms allow development of applications with explicit trusted and untrusted components, where trusted components run within the enclaves — enclaves tend to be small, separated components of an application, and therefore more amenable to formal analysis. Regardless of the platform used to implement enclave programs, we need techniques for writing safe programs. This dissertation studies the problem of developing enclave programs with provable guarantees (e.g. confidentiality of sensitive data), and because enclaves are the only trusted components, we achieve our goal of building trustworthy cloud services with a tiny software TCB.

**Verified Applications** Checking implementation code for safety is also a well studied problem. In order to guard against attacks that violate confidentiality properties, the application can be developed in a memory-safe language with information flow control, such as Jif [86]. We refer the reader to an existing comprehensive survey on the topic (e.g., [100]). Type systems proposed by Sabelfeld et al. [101], Barthe et al. [19], and Volpano et al. [121] enable the programmer to annotate variables that hold secret values, and ensure that these values do not leak. Balliu et al. [13] automate information flow analysis of ARMv7 machine code. Languages and verification techniques also exist for quantitative information flow (e.g., [57]). However, these techniques assume that the privileged software infrastructure on which the application runs is safe, which is unrealistic due to the large number of attacks observed in the wild. Our approach builds upon this body of work, showing how it can be adapted to the setting where programs run on an adversarial software infrastructure, and instead rely on trusted hardware for information-flow security. Furthermore, extending traditional type systems to enclave programs is non-trivial because the analysis must faithfully model the semantics of the trusted hardware primitives, which is another problem that we address in this thesis.

Complementary to the information flow analysis and program verification techniques, secure compilation is an approach for producing programs that are correct by construction. Patrignani et al. [95] develop abstract trace semantics for low-level code (not including side-channels) in order to build secure compilers for enclave platforms. In another work, Patrignani et al. [94] develop a fully abstract compiler (i.e., it forbids attacks on the target program that cannot be modeled at the level of the source program) from an object-oriented language to untyped assembly, which can be run within an enclave. While this line of work is closely related to this dissertation, there are important differences that make the two contributions complementary. First, instead of building a compiler from the ground up, we allow developers to use mainstream languages (such as C/C++) and production-grade compilers (such as gcc and Visual Studio) in our solution [112, 111], with the aim of facilitating practical deployment. Second, their secure compilation scheme assumes the safety of the underlying enclave platform. On the other hand, we formalize a set of properties that an enclave platform must satisfy, and prove these properties on models of the enclave platforms [116], thus extending the guarantees to lower abstraction levels — our model of the Sanctum [38] enclave platform serves as the reference specification of its primitives. That being said, we contend that both sets of techniques will find use in enclave programming in practice, and the choice would depend on factors such as the threat model, TCB assumptions, and developer constraints.

There are research proposals that offer stronger confidentiality properties, even preventing leaks via certain side channels. Oblivious RAM (ORAM) [54] protects against side channel leaks via the program's memory access patterns. Liu et al. [72] formalize memory trace obliviousness, and develop a compiler for producing memory trace oblivious programs by partitioning code and data across multiple ORAM banks for efficiency. GhostRider [70] presents a co-designed compiler and hardware ORAM for memory trace oblivious execution. However, these defenses assume the presence of special hardware, such as an ORAM

controller, and instead study how to use this special hardware in an efficient way. In this dissertation, we study defenses against side channels (specifically page-level memory access patterns) without assuming any special hardware support, which enable us to protect applications running on commodity hardware.

## 1.4 Thesis Organization

We first include some background material in Chapter 2, where we introduce the concept of enclaves, their use cases, and challenges in using enclaves for developing trustworthy cloud services. The remainder of the dissertation has two main parts.

**Trusted Platforms: Modeling and Verification** In Part I, we formalize an enclave program, the enclave platform, and various models of a privileged software adversary with varying capabilities. In Chapter 3, we present a formal model of enclave programs and their execution. In Chapter 4, we present a formal model of the enclave platform and the privileged software adversaries. We also define an abstract trusted platform whose operations simulate the primitives offered by popular enclave platforms such as SGX and Sanctum. The content of these chapters is based in part on joint work with Sriram Rajamani, Kapil Vaswani, and Sanjit Seshia [112], and on a separate joint effort with Pramod Subramanyan, Ilia Lebedev, Srini Devadas, and Sanjit Seshia [116]. Finally, in Chapter 5, we use the formal models from Chapter 3 and Chapter 4 to establish a theorem that any enclave program enjoys *secure remote execution* (defined in Chapter 5) when run on the abstract platform (and consequently, when run on either SGX or Sanctum). The work in this chapter is based on aforementioned effort with Pramod Subramanyan, Ilia Lebedev, Srini Devadas, and Sanjit Seshia [116].

**Secure Enclaves: Design and Verification** Part II of the dissertation develops a novel methodology to program enclaves that can be certified to not leak secrets, both via explicit outputs and via specific side channels. Chapter 7 presents a methodology and automatic tool, called Moat, to find vulnerabilities in enclave program binaries that cause it to output secrets to unprotected memory. Moat was produced as a result of a joint effort with Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani [112]. To address the scalability concerns of Moat, we develop /Confidential, presented in Chapter 8. We apply /Confidential to several practical enclave programs, including several Map-Reduce programs and SPEC-CPU benchmarks. The work in this chapter is based on joint work with Manuel Costa, Akash Lal, Nuno Lopes, Sriram Rajamani, Kapil Vaswani, and Sanjit Seshia [111]. In Chapter 9, we propose a novel technique and a toolchain consisting of a compiler and a verifier to produce safe enclave programs that do not leak secrets via their page-level memory access patterns. This chapter is based on joint work with Sriram Rajamani and Sanjit Seshia [113]. Finally, we conclude this dissertation and propose directions for future work in Chapter 10.

# Chapter 2

# Background

This chapter introduces the reader to the threat model addressed by this thesis, the primitives offered by enclave platforms, sample applications of enclaves, and the potential challenges that a developer faces while writing safe enclave programs. First, we motivate the use of trusted hardware primitives by describing potential attacks on the application once the attacker has compromised the privileged software layers e.g. OS and Hypervisor — we refer to an attacker that has compromised these privileged computing layers as a privileged software adversary. Next, we introduce the notion of an enclave program, and describe the trusted primitives offered by popular enclave platforms such as Intel SGX and RISC-V Sanctum. We also demonstrate typical use cases of enclaves via some sample applications: a one time password service, and a map-reduce framework for processing sensitive data. We conclude this chapter by describing the challenges in programming enclaves with provable safety guarantees in the presence of a privileged software adversary.

## 2.1 Attacks from a Privileged Software Adversary

We consider the adverse setting where powerful attackers have taken control of the cloud's entire software infrastructure — including the privileged software layers (e.g. OS, Hypervisor, BIOS firmware) that software services rely upon — on any number of machines in the cloud. These attacks can be carried out by malicious datacenter insiders and remote attackers, who exploit vulnerabilities in the large software trusted computing base of such systems — modern operating systems and hypervisors amass several millions of code [99], and many critical vulnerabilities have been demonstrated in these systems [33, 63]. However, we assume the presence of trusted platforms (e.g., Intel SGX processors), which the attacker is unable to physically tamper; we describe these trusted platforms and their primitives in Section 2.2.

Such a privileged software adversary controls the entire software stack on the machine, which allows it to read or modify all I/O interaction. The adversary may record, replay, and modify all I/O events, which include network packets and files written to persistent storage. As a consequence, the attacker may perform denial-of-service, and defending against such

attacks is beyond the scope of this work. The attacker also controls the I/O peripherals and can program them to generate arbitrary I/O events.

By compromising the privileged software layers, the adversary also gets to read or modify the application's memory, which may contain sensitive code and data. Therefore, applications running on a compromised OS do not enjoy any confidentiality or integrity guarantees. The goal of using trusted enclave platforms is to prevent confidentiality and integrity violations on the code and data of an application.

## 2.2 Enclaves using Trusted Hardware Primitives

Processor vendors have started to support hardware-based *containers* (such as Intel SGX enclaves [61], RISC-V Sanctum enclaves [38], and ARM TrustZone trustlets [9]) for isolating sensitive code and data from hostile or compromised hosts.

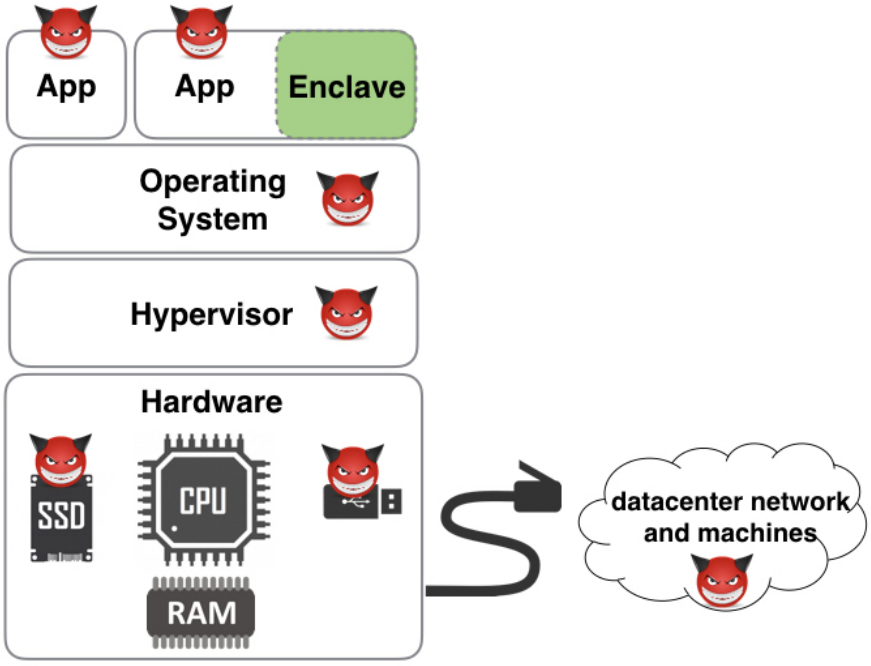



*Figure 2.1*: Threat Model: The adversary controls all privileged software layers on the platform, and all hardware units except the CPU and memory. The enclave is the only trusted software component.

An *enclave platform* implements primitives to create protected memory regions, called *enclaves*, that contain both code and data and are isolated from all other software in the system. The processor monitors all accesses to the enclave: only code running in the enclave can access the enclave's memory. As an example, Intel's SGX instructions enable the creation of user-mode enclaves in the hosting application's address space. The enclave platform

allows the enclave program to access the entire address space of the hosting application. This enables efficient I/O interaction with the external world — system calls are disabled in enclave mode because the OS is untrusted. The external world can only transfer control to the enclave at statically-defined locations called *entrypoints*. In addition to enabling isolated execution, the enclave platform implements primitives for generating attested statements: code inside an enclave can get messages signed using a per-processor private key along with a hash-based measurement of the enclave. This allows other trusted entities to verify that messages originated from the desired enclave running on a genuine platform. Finally, we assume the enclave platform implements a cryptographically secure random number generator which the enclave can use for cryptographic operations such as key generation.

Primitives to create enclaves can also be provided by hypervisors [60, 34, 127], with the caveat that the hypervisor becomes part of the TCB, but in this thesis we assume enclaves are provided directly by the processor. Regardless of the infrastructure used to implement enclaves, we require that an enclave program enjoys the following guarantees and primitives:

**Isolated Execution** From its creation until teardown, an enclave's memory and registers are protected from reads or writes by all untrusted software: other enclaves, host application, OS, hypervisor, system management code, etc. The only way an attacker affects the enclave's execution is by providing inputs, but this is of the enclave's own volition as the enclave performs the reads from the unprotected memory (host application's address space) to fetch inputs. Enclaves cannot invoke system calls to perform I/O with the external world because the OS cannot be trusted to modify the enclave's memory, and hence, the enclave proxies all I/O interactions via the untrusted host application by reading and writing to its memory. Note that since inputs are untrusted and outputs are observed, the enclave must implement necessary checks to protect integrity and confidentiality. Furthermore, the trusted platform does not promise performance guarantees, and in fact, does not protect from denial of service attacks — at any point, the CPU may asynchronously exit from the enclave, but it takes necessary measures to protect the enclave's state.

**Attested Statements** Since the enclaves are launched by the untrusted code (by invoking trusted primitives), an attacker controls the initial code and data of the enclave, whose binary is typically stored in untrusted persistent storage or unprotected memory. Consequently, to prove to a remote user that she is communicating with the expected enclave, the hardware platform computes a hash digest (a.k.a. measurement) of the enclave while it is being launched, which forms the cryptographic identity of the enclave — the hash is computed over all components of the initial state that affects execution, including code and data memory, page table permissions, etc. On invocation of the attestation primitive, the hardware signs the message along with the identity, thus allowing a remote user to verify that the message originated from an enclave with the expected identity which is running on a genuine trusted platform [93]. Enclaves typically use this primitive to authenticate messages of the initial key-exchange protocol to establish secure channels with remote entities.

**Randomness** Enclaves require a hardware-based source of entropy for random bits (e.g. rdrand instruction on Intel CPUs) in order to perform various cryptographic operations.

### 2.2.1 Intel SGX Enclaves

The SGX instructions allow an OS to create an enclave containing code and data, on behalf of a user-level *host application*. The enclave's memory resides within the untrusted host application's virtual address space, but is protected from accesses by that host application or any privileged software — only the enclave code is allowed to access enclave memory. Furthermore, to protect against physical attacks on the RAM, the SGX processor also encrypts (using authenticated encryption) the data stored to the RAM chip.

The OS creates an enclave using a combination of instructions: `ecreate`, `eadd`, `eextend`, and `einit`. The OS invokes `ecreate` to reserve protected memory for enclave use. To populate the enclave with code and data, the host application uses a sequence of `eadd` and `eextend` instructions. `eadd` loads code and data pages from non-enclave memory to the enclave's reserved memory. `eextend` extends the current enclave measurement with the measurement of the newly added page. Finally, `einit` terminates the initialization phase, which prevents any further modification to the enclave state (and measurement) from non-enclave code. The host application transfers control to the enclave by invoking `eenter`, which targets a programmer defined entry point inside the enclave (via a callgate-like mechanism). The enclave executes until one of the following events occur: (1) enclave code invokes `eexit` to transfer control to the host application, (2) enclave code incurs a fault or exception (e.g. page fault, divide by 0 exception, etc.), or (3) the CPU receives a hardware interrupt and transfers control to a privileged interrupt handler. In the case of faults, exceptions, and interrupts, the CPU saves state (registers, etc.) in State Save Area (SSA) pages within enclave memory, and can later resume the enclave in the same state by invoking the `eresume` instruction. Although a compromised OS may use this design to launch denial of service attacks, an enclave's private state remains inaccessible to the attacker.

The reader may have observed that before enclave initialization, code and data is open to eavesdropping and tampering by adversaries. For instance, an adversary may modify the enclave's binary such that it contains a vulnerability that leaks a user's login credentials. SGX provides an attestation primitive called `ereport` to defend against this class of attacks. The enclave program performs remote attestation by invoking `ereport` on a chosen message, which generates a hardware-signed report of the message along with the enclave's measurement. The enclave program sends this report to the remote verifier, who can then check that the message originated within an enclave with the expected measurement, running on a genuine SGX CPU. The enclave can use `egetkey` to attain a hardware-generated sealing key which is unique to that enclave (and the particular CPU), and store secrets to untrusted storage by encrypting them with the sealing key.

### 2.2.2 RISC-V Sanctum Enclaves

In contrast to SGX, Sanctum's primitives are implemented using a combination of software (that executes at the highest privilege level), called the monitor, and minimal hardware extensions to the RISC-V processor architecture. That being said, it presents a similar programming model as SGX, and also guards various forms of side channel leaks that SGX does not defend against, e.g., cache timing leaks, revelation of faults and exceptions to the OS, page-level memory access patterns, etc. We refer the reader to Section 4, Table 12 of [37] for a detailed comparison between SGX, Sanctum, ARM TrustZone, and other trusted execution platforms.

Similar to SGX, Sanctum enables an OS to launch user-mode enclaves, on the behalf of the hosting application, by invoking a sequence of monitor API calls. The OS invokes create_enclave to commence the enclave launch, which reserves a metadata structure within Sanctum's private state — the metadata structure is used throughout the lifetime of the enclave for implementing various security checks. Next, the OS commits a subset of physical memory by calling assign_dram_region, and a set of threads by calling load_thread. The OS is also responsible for setting up the page table translation by invoking load_page_table — the enclave's page tables are placed within the enclave's memory to avoid tampering by the OS and leaking memory access patterns via the accessed and dirty bits in page table entries. copy_page is used to load pages from non-enclave memory to enclave memory during the launch process. Once the launch process is completed, the OS invokes init_enclave to to prevent further modifications by the untrusted code. Execution alternates between the enclave and the untrusted code; the OS enters the enclave at a pre-configured entrypoint using enter_enclave, and the enclave exits by calling exit_enclave. The enclave can request attested statements by communicating with a special quoting enclave (using send_message and read_message API). The OS can choose to teardown the enclave by sending interprocessor interrupts to exit out of the enclave threads, flushing the translation lookaside buffers, and invoking delete_enclave — at this point, the monitor claims all physical memory (allocated by assign_dram_region) and zeroes them out.

**Side Channel Defenses** The key contributions of Sanctum, relative to SGX, are its mechanisms for preventing certain classes of side channel leaks. Sanctum shares resources (e.g. memory, cache, TLB, branch target buffers) amongst enclaves and the OS either in time or in space, but never both. For instance, a CPU core's cache is flushed on control transfers between the OS and the enclave, whereas the last level cache is partitioned such that an enclave cannot target the same cache set as another enclave or the OS — this prevents the attacker from learning an enclave's memory access pattern by observing cache hits or misses. As another example of side channel defense, the monitor does not notify the OS on a fault or exception generated during an enclave's execution, which prevent revealing page-level access pattern (from page faults). This side channel was recently exploited [125] to extract significant amount of sensitive data from Intel SGX enclaves.

## 2.3 Sample Applications of Enclaves

### 2.3.1 One Time Password Service

We demonstrate the use of Intel SGX primitives using an example of a one-time password (OTP) service, although the exposition extends naturally to other trusted platforms such as RISC-V Sanctum. OTP is typically used in two factor authentication as an additional step to traditional knowledge based authentication via username and passphrase. A user demonstrates ownership of a pre-shared secret by providing a fresh, one-time password that is derived deterministically from that secret. For instance, RSA SecurID® is a hardware-based OTP solution, where possession of a tamper-resistant hardware token is required during login. In this scheme, a pre-shared secret is established between the OTP service and the hardware token. From then on, both the authentication server and the token compute a fresh one-time password (OTP) as a function of the pre-shared secret and time duration since the secret was provisioned to the token. The user must provide the OTP value displayed on the token during authentication, in addition to her username and passphrase, which the server compares to the expected value. This OTP scheme is both expensive and inconvenient because it requires distributing tamper-resistant hardware tokens physically to the users. Although pure software implementations have been attempted, they are often prone to infrastructure attacks from privileged OS-level malware, making such solutions untrustworthy.

The necessary primitives for implementing this protocol securely are (1) ability to perform the cryptographic operations (or any trusted computation) without interference from the adversary, (2) protected memory for computing and storing secrets, (3) root of trust for measurement and attestation, and (4) a secure entropy source for generating Diffie-Hellman parameters during secure channel establishment between the user's client and the authentication server. Intel SGX and RISC-V Sanctum processors provide all of these primitives. Hoekstra et al. [59] propose the following OTP scheme based on SGX, which we implement (shown in Figure 2.2) — in Chapter 7, we formally verify this implementation (compiled binary) to have confidentiality properties, i.e., we prove that it does not leak secret state to adversary-visible memory in any execution. The scheme involves two enclaves: a setup enclave that receives the pre-shared secret from the server and saves it securely to persistent storage for future use, and an authentication enclave that reads the saved pre-shared secret from persistent storage and communicates to the server to perform authentication. We only discuss the setup enclave here, though most concepts apply to the authentication enclave as well, and other enclaves in general.

In the following protocol, an OTP server provisions the pre-shared secret to a client, which is running on a SGX CPU and a potentially malicious OS. This involves the following steps:

0. The host application on the client sets up an enclave (using `ecreate`, `eadd`, `eextend`, and `einit` instructions) that contains trusted code for the client side of the protocol.

1. The client and OTP server establish establish a TLS-like secure channel. They engage in

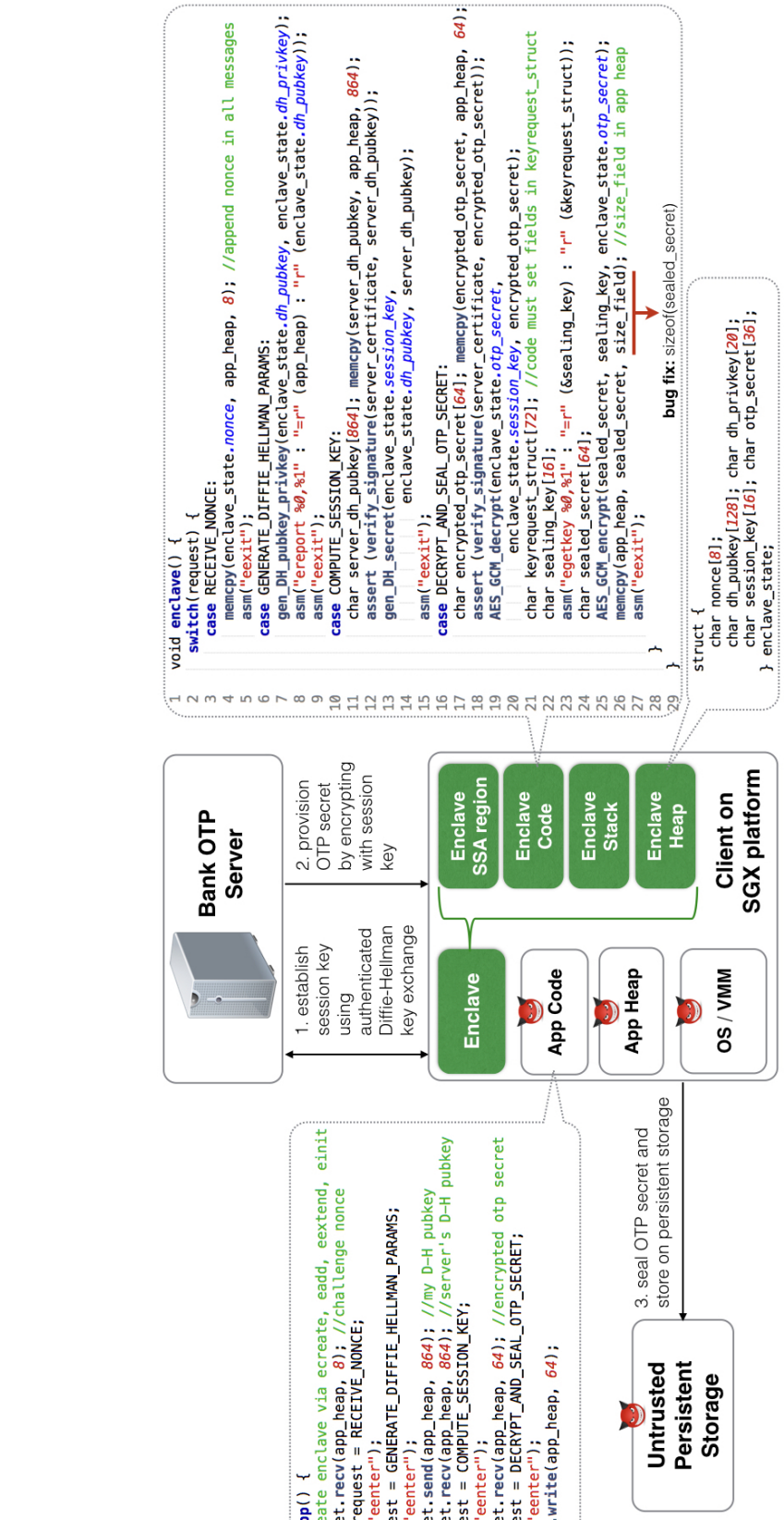



*Figure 2.2*: Running OTP Example. The enclave performs trusted cryptographic operations, and the host application performs untrusted tasks such as UI handling and network communications with the OTP server.

an ephemeral Diffie-Hellman key exchange in order to establish a symmetric session_key. To prevent man-in-the-middle attacks, the client authenticates all messages during the key exchange via the use of `ereport` instruction — the messages produced by `ereport` contain a signature over the Diffie-Hellman public key dh_pubkey and the enclave's measurement. The signature guarantees that the report was generated by an enclave on an Intel SGX CPU, while the measurement guarantees that the reporting enclave was not tampered during initialization. The server signs its messages using a RSA private key, with the corresponding public key available within a certificate. After verifying the signatures, both the client and OTP server compute the symmetric session_key.

2. The OTP server sends the pre-shared OTP secret to the client by encrypting it with the session_key using the AES-GCM [102] cipher, which implements authenticated encryption with associated data. The client decrypts the message using session_key to retrieve the pre-shared otp_secret.

3. For future use during two-factor authentication, the client requests sealing_key (using `egetkey` instruction), encrypts otp_secret using sealing_key, and writes the sealed_secret to persistent storage.

An application might use enclave code to implement trusted computation such as the cryptographic operations, stores secrets in the enclave heap, and uses non-enclave code (host application, OS, VMM, etc.) for untrusted computation such as I/O. SGX prevents the enclave code from invoking any privileged instructions such as system calls, thus forcing the enclave to rely on non-enclave code to issue system calls, perform I/O, etc. SGX allows the enclave to access the entire address space of the host application for efficient I/O, which allows the enclave code to pass messages to/from the non-enclave code. For instance, to send the Diffie-Hellman public key to the server, the enclave (1) invokes `ereport` with enclave_state.dh_pubkey, (2) copies the report to the unprotected host application's memory app_heap, (3) invokes `eexit` to transfer control to the host app, and (4) waits for app to invoke the socket system calls to send the report to the bank server. Over their lifetimes, app and enclave perform several `eenter` and `eexit`, thus alternating between trusted and untrusted computation.

### 2.3.2 VC3: Trustworthy Data Analytics using SGX

VC3 [107] is a map-reduce framework for computing on sensitive data within SGX enclaves, whose architecture can be summarized in Figure 2.3. In VC3, only the map and reduce functions are executed within enclaves, whereas the rest of the large Hadoop stack is untrusted and only manages encrypted data. The map and reduce functions receive encrypted input from the untrusted Hadoop layer, decrypt the input within the enclave's private memory, process it, and send the encrypted result (e.g., list of key-value pairs to be sent to the reducer) back to Hadoop.

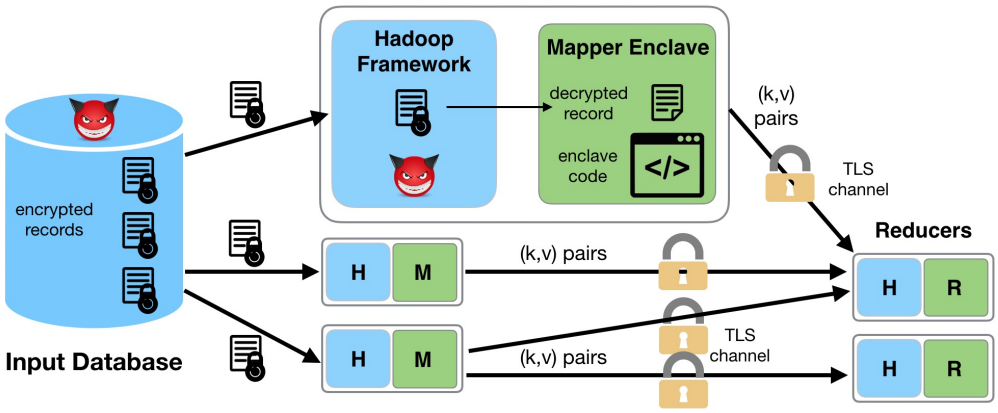



*Figure 2.3*: Map-Reduce computation in VC3. The Hadoop layer only accesses encrypted data, and communication between enclaves is encrypted via TLS. This figure is derived from Figure 3 of [107].

We illustrate the use of enclaves in hardening a sample Map-Reduce program that computes on sensitive user data, whose reducer function is shown in Figure 2.4. We show a simple wordcount application that counts the number of occurrences of distinct words in a sensitive document. The reducer method (implemented within an SGX enclave) fetches encrypted key-value pairs from the untrusted Hadoop's memory, decrypts them, computes a sum of all the input values, and sends the encrypted result to the user via the untrusted Hadoop platform. Consider the following attack scenarios on this program. First, without SGX-enabled protections, any kernel-level malware can extract the decrypted key-value pairs in the reducer's memory; hardware-based isolation guarantees of SGX prevents this attack. Second, prior to launching the Map-Reduce program, the attacker may replace the reducer's binary (which was uploaded by the user) to a buggy version that emits all the secret data to the attacker; SGX's measurement and attestation primitives enable the user to verify that the enclave binary is equal to the one that the user expects. Finally, the enclave may encrypt and store secrets to disk, and SGX's sealing primitive guarantees that only the intended enclave is able to decrypt them. The primitives of enclave platforms (such as SGX and Sanctum) are necessary for secure processing of sensitive data, amongst other security-critical applications.

## 2.4 Challenges of Trusted Computing using Enclaves

The primitives offered by trusted enclave platforms — isolated execution, attested statements, and a secure entropy source — are necessary for any form of trustworthy computing on sensitive data, but not sufficient by themselves. The enclave programs must also be secure, which includes, at the very least, a provable guarantee that sensitive data is never leaked to the attacker in any execution, under any potential attack scenario — we call this property confidentiality. However, guaranteeing that enclave programs satisfy confidentiality

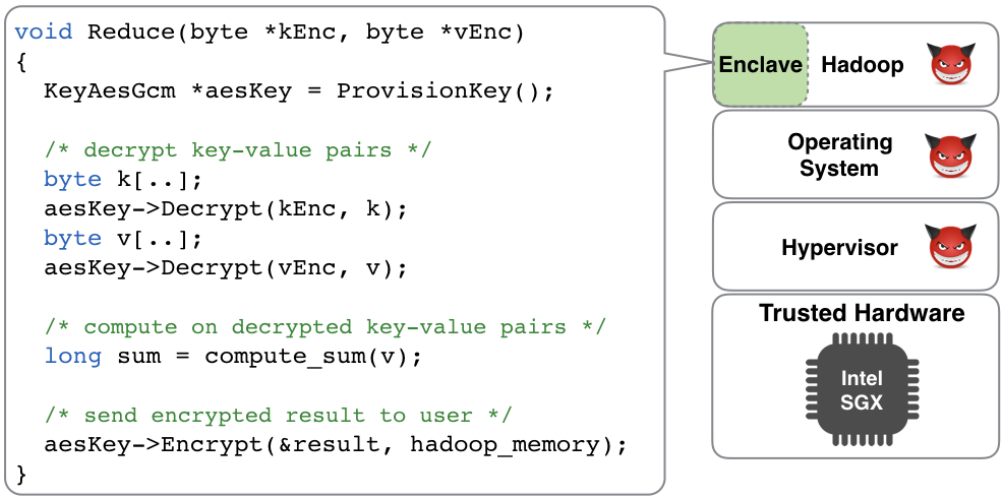



*Figure 2.4*: Executed within an enclave, this reducer method computes on sensitive cleartext data.

has the following challenges.

**Formal Model of Enclave Programs, Platforms, and Adversaries** Proving confidentiality requires precise reasoning of all potential executions of the enclave program in the presence of a privileged software adversary, which requires formalization of the following aspects:

- Enclave platform's primitives, e.g., the semantics of x86 and SGX instructions — this forms the contract between the enclave developer and the underlying enclave platform. Furthermore, due to the large diversity in trusted execution environments available on mainstream processors, one challenge is developing a formal framework that allows us to reason about enclave execution generally across a wide variety of platforms; this would also allow us to develop enclave programs that port seamlessly across different platforms.

- Models of the privileged software adversaries, i.e., the set of allowed operations and observations. A formal adversary model enables us to establish theorems about security properties offered to enclave programs on enclave platforms with privileged software adversaries.

**Preventing Explicit Leaks via Outputs** Like any other application, to serve a useful purpose, an enclave must interact with the external world to receive inputs and produce outputs, and this gives the adversary a mechanism to interact with the enclave and exploit any of its vulnerabilities.

For instance, since the enclave fetches its inputs by reading from non-enclave memory, the attacker can modify non-enclave memory at any time and perform time-of-check-to-time-

```
...
/* size_field stored in a struct in
     non-enclave memory */
memcpy(app_heap, sealed_secret, size_field);
...
```

```
long compute_sum(char *v) {
  ...
  /* sprintf without any size checks */
  sprintf(log, "%s", v); //perform logging
  ...
}
```

```
if (secret_age > 65) {
  seniorsCount++;
  specialComputation();
} else {
  normalComputation();
}
```

*Figure 2.5*: Explicit leaks from memory safety errors (left) and implicit leaks from side channels (right)

of-use style attacks on an insecure enclave. In fact, the enclave code in Figure 2.2 (relevant snippet repeated in Figure 2.5) has such a vulnerability: the enclave copies encrypted data from enclave memory to non-enclave memory, but the size of the data copied is determined by a variable size_field, which resides in a struct in non-enclave memory. Thus, by manipulating the value of this variable the adversary can trick the enclave code into leaking secrets to non-enclave memory. To defend against such class of attacks, the enclave should copy the inputs to protected memory before computing on them. Furthermore, since the attacker can access any outputs written to non-enclave memory, the enclaves should encrypt secrets before writing them to non-enclave memory. For instance, the enclave code in Figure 2.4 (relevant snippet repeated in Figure 2.5) could be vulnerable if the compute_sum procedure has a memory safety bug that overwrites the part of memory holding the cryptographic key aesKey, which could be modified to an attacker-chosen value or leaked to non-enclave memory, thus breaking the semantic security assumed of the AES encryption. As shown in Figure 2.5, the buggy code may invoke sprintf without checking that the output buffer log is sufficiently large; this causes sprintf to overwrite memory beyond log, which may include aesKey, or even a code pointer (e.g. return address) on the program's stack which leads to control flow integrity violations [117] allowing the attacker to run arbitrary code within the enclave. Writing safe enclaves that avoid such attacks is non-trivial, and prior experience suggests that non-experts make similar errors as the aforementioned vulnerabilities — memory safety bugs in software written in low-level languages such as C/C++ are one of the oldest and most widespread problems in computer security [117].

**Preventing Implicit Leaks via Side Channels** To add to the developer's woes, many mainstream trusted platforms do not offer idealized execution guarantees. While they may offer isolated execution, they do not necessarily offer strict confidentiality. Due to various performance-related reasons, platforms, such as Intel SGX, do end up leaking information about the enclave's execution via various software and hardware side channels (e.g. cache

timing attacks, memory access patterns), and preventing these leaks requires even greater understanding of hardware, careful programming practices, and tool support for automatically enforcing and verifying defenses. Consider the enclave snippet in Figure 2.5 where a secret state (or input) is used to conditionally execute code within the if branch. Assume that the adversary can observe all the page-level accesses made by the enclave to code and data pages — we discuss in Chapter 9 how an adversary can leverage the compromised OS on a SGX platform to observe this side channel information. Based on value of sensitive state, if the user's age is above 65, the attacker observes an access to the data page holding the seniorsCount variable, which is not performed if the user's age is below 65, thus allowing the adversary to infer the outcome of the branch condition. Furthermore, if the x86 instructions implementing the specialComputation procedure are placed on a separate page than the instructions implementing the normalComputation procedure, or if the procedures have different number of instructions, then the adversary can infer the secret branch condition. Chapter 9 contains other examples of such leaks, which are non-trivial to defend against, and require automatic tool support.

Software defenses against side channels are still a very active area of research, as typical solutions either provide insufficient defenses, offer limited expressiveness, cause severe performance penalties, and offer limited usability for non-expert developers. This dissertation presents a novel methodology for safely programming enclaves and formally certifying the absence of vulnerabilities that leak secrets, both via explicit outputs and certain side channels.

# Part I

# Trusted Platforms: Modeling and Verification

# Foreword on Part I

Despite growing interest, there has only been informal security analysis of enclave programs, platforms, and threat models. This lack of formalization has several consequences. Developers of enclave programs cannot formally reason about security of their programs: incorrect use of hardware primitives or informal characterization of attacker's abilities lead to vulnerabilities in the enclave program. Furthermore, hardware designers cannot formally state security properties of their architectures and are unable to reason about potential vulnerabilities of enclaves running on their hardware. This part of the thesis bridges these gaps by developing a formal framework to reason about an enclave's execution in the presence of a privileged software adversary.

Specifically, we present a formal model of enclave programs and their execution (Chapter 3), and a formal model of the enclave platform and the privileged software adversaries (Chapter 4). Furthermore, to generalize our guarantees beyond a specific platform, we propose an abstract trusted platform (Chapter 4) whose operations simulate the primitives offered by popular platforms such as SGX and Sanctum. We then use these formal models to establish a theorem (Chapter 5) that any enclave program enjoys a set of security properties when run on the abstract platform (and consequently, SGX, Sanctum, etc.), thus enabling us to reason about enclave executions on any number of enclave platforms in the presence of a privileged adversary. The contributions in this part of the thesis are vital to our goal of certifying confidentiality properties of enclave software with a tiny trusted computing base.

# Chapter 3

# Formal Semantics of Enclave Execution

In this chapter, we define a formal semantics of enclave execution, which includes: 1) a characterization of the enclave's state, inputs and outputs; 2) formalization of the syntax and semantics of enclave programs; and 3) a model of computation for enclave programs. Not surprisingly, the model of enclave computation assumes certain properties of the enclave platform, e.g., isolation of the enclave's private state from privileged software layers, absence of undocumented side channels that reveal enclave's private state, etc. Therefore, we specify the expected guarantee that an enclave expects from the enclave platform in Chapter 5, termed *secure remote execution*, but in this chapter we assume a trusted enclave platform and define the enclave's semantics based on this assumption. The formal framework developed in this chapter enables the developer (or user) to precisely define the expected runtime behavior of an enclave in the presence of a privileged software adversary — when the user outsources an enclave to a remote platform, she seeks a guarantee that the enclave be executed according to the expected behavior.

In the remainder of this chapter, we define an enclave's representation as provisioned by the user (Section 3.1), enclave's state (Section 3.2), inputs and outputs (Section 3.3), syntax and semantics of enclave code (Section 3.4 and Section 3.5), and the model of computation within enclaves (Section 3.6).

## 3.1 Enclave Program Representation

An *enclave platform* implements primitives to create protected memory regions, called *enclaves*, that contain both code and data and are isolated from all other software in the system. To outsource the enclave's execution, the user sends the enclave program to a remote enclave platform over an untrusted channel (Figure 3.1). The untrusted OS invokes the enclave platform's primitives to launch an enclave containing the program. While running, an enclave may invoke the enclave platform's primitives to get attested statements and random bits. The enclave may also send outputs to the user by proxying them via the host application's unprotected memory.

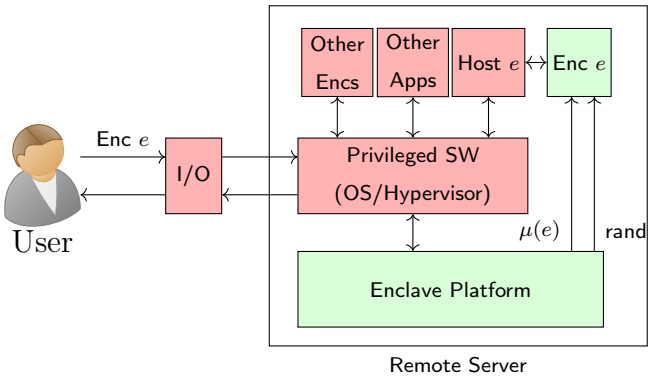



*Figure 3.1*: Outsourced Execution of Enclave Program.

At the time of launch, an enclave is assigned a protected address range of virtual memory containing a set of code pages, which holds the enclave's user-mode instructions, and a set of data pages, which can be used as enclave's private heap and stack space. The user specifies the expected initial state of enclave's memory, i.e., the value stored at each virtual address within this range, at the time of launch. The enclave's configuration also includes (1) the entrypoint address, which is the target of any control transfer from non-enclave mode into this enclave, (2) the virtual address range *evrange* which maps the enclave's protected memory (i.e., each address in *evrange* either maps to a physical address that is owned by the enclave or is inaccessible), and (3) permissions for each address within *evrange*. The initial state of enclave memory and the aforementioned configuration forms the unique identity (or representation) of the enclave, and must be provided by the user in order for the enclave platform to launch the enclave. Note that since the attacker may modify the enclave's identity prior to launch, the enclave platform includes a hash-based measurement of this identity in all attested statements, thus enabling the user to verify that the enclave (that produced the attested statement) was launched with the expected initial state and configuration.

More formally, the user (in Figure 3.1) ships an enclave $e = (init_e, config_e)$ — $e$ is the enclave's unique identity or representation. $init_e$ specifies the enclave's initial state of memory at launch time, and it specifies a value at each address within the protected range $config_e.evrange$. The configuration $config_e$ defines the enclave's entrypoint $config_e.entrypoint$, its virtual address range $config_e.evrange$, and its access permissions $config_e.acl$.

## 3.2 Enclave's State

Like any program, an enclave maintains some private state (inaccessible to the privileged software adversary), receives inputs and sends outputs (both accessible by the privileged adversary), and computes by invoking machine instructions (e.g., user-mode x86 instructions). At any point of time, the enclave platform is in some state $\sigma$, which we leave abstract here

because different enclave platforms (e.g., SGX and Sanctum) will maintain different types of state. That being said, we can concretely define an enclave's private state, which all enclave platforms are required to provide to an enclave. The enclave's state $E_e(\sigma)$ is a projection function of the platform's state $\sigma$, and it contains a valuation of the following state variables: (1) enclave memory $\texttt{vmem} : \text{VA} \to \text{W}$ which is a partial map from virtual addresses within $\mathit{config}_e.\mathit{evrange}$ to machine words, (2) a set of general-purpose registers $\texttt{regs} : \mathbb{N} \to \text{W}$ and status flags $\texttt{flags} : \mathbb{N} \to \mathbb{B}$, which are indexed by a natural number, (3) program counter $\texttt{pc} : \text{VA}$, which is a virtual address within $\mathit{config}_e.\mathit{evrange}$, and (4) configuration $\mathit{config}_e$, which is copied verbatim from $e$ and remains constant throughout the enclave's execution. Each of the aforementioned components of an enclave's state remains inaccessible to the untrusted privileged software during an enclave's execution — this is assumed in our definition of enclave execution, and it is implied by the platform's safety guarantees (defined in Chapter 5). Recall that $\mathit{init}_e$ specifies the state of enclave's memory (vmem) at the time of launch. We abuse notation slightly to also write $\mathit{init}_e(E_e(\sigma))$ to mean that $E_e(\sigma)$ is in its launch state, i.e., the value of vmem in $E_e(\sigma)$ is $\mathit{init}_e$.

## 3.3 Enclave's Inputs and Outputs

In addition to the private state $E_e(\sigma)$, an enclave accesses non-enclave memory for reading inputs and writing outputs, thus allowing the untrusted software to control their values — since addresses outside $\mathit{config}_e.\mathit{evrange}$ can be read and modified by both the adversary and the enclave $e$, we assume reads performed by $e$ from these addresses return unconstrained values. The enclave may also invoke the enclave platform's primitive to get random numbers, which we also treat as inputs to $e$. Therefore, we define enclave's input in state $\sigma$ to be $I_e(\sigma) \doteq \langle I_e^R(\sigma), I_e^U(\sigma)\rangle$, where $I_e^R(\sigma)$ denotes the random input and $I_e^U(\sigma)$ denotes the contents of untrusted memory (which the untrusted OS may read and write). $I_e^R(\sigma)$ is the random number that is provided by the platform while executing the instruction in state $\sigma$ ($\epsilon$ if randomness was not requested in that instruction). $I_e^U(\sigma)$ is an evaluation to non-enclave memory, which is a partial map from virtual addresses (outside $\mathit{config}_e.\mathit{evrange}$) to machine words. Similarly, we define enclave's output $O_e(\sigma)$ to be a projection of the machine state that $e$ may write and the untrusted privileged software may read; specifically, $O_e(\sigma)$ specifies an evaluation to non-enclave memory, which is a partial map from virtual addresses (outside $\mathit{config}_e.\mathit{evrange}$) to words. Note that a privileged software attacker runs concurrently with $e$, and therefore maintains a component of state $\sigma$ that is inaccessible to $e$.

## 3.4 Syntax of Enclave Code

The enclave computes by invoking user-mode instructions for performing bitvector operations on registers, memory accesses, and invoking primitives for generating attested statements, randomness, and exiting enclave mode. These instructions may read enclave's private

$$
\begin{array}{llll}
\mathsf{v} \in \textit{Vars} & ::= & \texttt{regs} \mid \texttt{flags} \mid \texttt{vmem} \mid \texttt{pc} \\
c \in \textit{Constants} & & \\
q \in \textit{Relations} & ::= & \mathsf{lt} \mid \mathsf{gt} \mid \mathsf{slt} \mid \mathsf{sgt} \mid \ldots \\
f \in \textit{Functions} & ::= & \mathsf{add} \mid \mathsf{xor} \mid \mathsf{extract} \mid \mathsf{concat} \mid \ldots \\
e \in \textit{Expr} & ::= & \mathsf{v} \mid c \mid f(e, \ldots, e) \mid q(e, \ldots, e) \\
\phi \in \textit{Formula} & ::= & \mathsf{true} \mid \mathsf{false} \mid e = e \mid q(e, \ldots, e) \mid \phi \wedge \phi \mid \neg\phi \\
i \in \textit{Instr} & ::= & \mathsf{store_n}(e, e) \mid \mathsf{v} := \mathsf{load_n}(e) \mid \mathsf{v} := e \mid \\
 & & \texttt{jmp}\ e \mid \mathsf{cjmp}(\phi, e, e) \mid \texttt{call}\ e \mid \mathsf{ret} \\
 & & \mathsf{exit} \mid \mathsf{attest}(e, e) \mid \mathsf{v} := \mathsf{rand}() \\
p \in \textit{Program} & ::= & i\ ;\ i
\end{array}
$$

*Figure 3.2*: Syntax of Enclave code.

state and inputs and may update the enclave's state and produce outputs. In this section, we formalize the syntax for each of enclave's allowed instructions.

To encode both CISC-like instructions in x86-64 and RISC instructions in RISC-V Sanctum, we define a RISC-like grammar as shown in Figure 3.2. Each machine code instruction in the enclave program is translated to a sequence of instructions (*Instr*) in the presented language. Translators exist for lifting x86-64 instructions (and RISC-V instructions) to this language, which bears strong resemblance to the BAP [31] intermediate language — the BAP framework implements a translation for x86-64 to BAP intermediate language, which we use with minor modifications [1]. As we show later in this dissertation, by using this simpler language, we simplify the implementation of enclave program verifiers and also reuse large parts of the verification toolchain across different trusted hardware platforms. The code of an enclave program, denoted $p$, is a sequence of instructions, with optional labels encoding the address of that instruction — since an x86 instruction is encoded using multiple instructions from *Instr*, not all instructions in $p$ are given an address.

These instructions cause updates to the enclave's state $E_e(\sigma)$. In Figure 3.2, we list the state variables *Vars* (containing `regs`, `flags`, and `vmem`), which are components of $E_e(\sigma)$ and are accessed by the instructions in *Instr*. Recall that `regs` are CPU registers (e.g., rax, rsp, etc.) that are 64-bit values in the case of x64, and `flags` are CPU flags (e.g., CF, ZF, etc.) that are boolean values. Virtual memory (`vmem`) is modeled as a map from 64-bit bit-vectors to 8-bit bit-vectors. The instruction pointer `pc` stores the address of the next instruction to be executed; although `pc` is a component of enclave state, it is updated automatically by the CPU and is therefore not listed explicitly in *Vars*.

Memory accesses are encoded using $\mathsf{load_n}$ and $\texttt{store}_\mathsf{n}$ functions, where n denotes the size of the data in bytes. Assignments can be one of following two forms: (1) $\mathsf{v} := e$ sets $\mathsf{v} \in \textit{Vars}$ to the value of expression $e$, which is used to encode bitvector operations on registers such as add, (2) $\mathsf{reg} := \mathsf{load_n}(e)$ sets $\mathsf{reg} \in \texttt{regs}$ to the value of the memory at the

[1] the translation assumes single thread execution, a form of control flow integrity (which we later verify to hold on the enclave code), and separation of code and data: executable memory is non-writable

address that $e$ evaluates to. Control flow is changed using `call`, ret, and `jmp` instructions, which override the value of `pc`. Both `call` and ret are semantically equivalent to the x64 call and return instructions, respectively — `call` pushes the return address on the stack, and the ret instruction pops the return address from the stack and jumps to the instruction at that address. `jmp` $e$ encodes an unconditional jump by setting `pc` to the value of $e$ in that platform's state; `cjmp`$(\phi, e, e)$ encodes conditional jumps.

Finally, an enclave may invoke primitives for generating attested statements, random bytes, and exiting to non-enclave code. attest$(e_1, e_2)$ takes as argument $e_1$ the starting address of a fixed-size buffer (e.g., 64 bytes), and writes the attested statement in a fixed-size buffer starting at address $e_2$. rand$(e)$ produces a random byte at address $e$. exit sets `pc` to some non-enclave address, and puts the CPU to non-enclave mode.

## 3.5 Semantics of Enclave Code

The enclave platform must guarantee an operational semantics for each instruction in *Instr*. The operational semantics serves as a formal contract between the user (or enclave developer) and the enclave platform, and we define the expected execution of an enclave program based on this semantics.

We formalize the operational semantics for the instructions (*Instr*) in Figure 3.3. Note that we only define the syntax and semantics for instructions executed in enclave mode — the platform's safety guarantee lets us assume that enclave's state is unmodified by any instruction executed in non-enclave mode. Let `instr`$(\sigma)$ be the instruction executed in state $\sigma$ (computed from the instruction pointer `pc` and the contents of `vmem`). The semantics of an instruction $i \in$ *Instr* is given by the relation $\Downarrow$, where $\langle i, \sigma \rangle \Downarrow \sigma'$ if and only if $i =$ `instr`$(\sigma)$ and there is an execution of $i$ starting at $\sigma$ and ending in $\sigma'$ (as per the operational semantics). The instructions update the platform's state $\sigma$, which in turn causes updates to enclave's state $E_e(\sigma)$ and output $O_e(\sigma)$ since they are projections of $\sigma$; recall that $\sigma$ contains a valuation of variables in *Vars*. By definition, an enclave cannot modify anything beyond its private state $E_e(\sigma)$ and output $O_e(\sigma)$. The operational semantics of an instruction ß $\in$ *Instr* is defined as part of the platform's transition relation $\rightsquigarrow$: $(\sigma_i, \sigma_j) \in \rightsquigarrow$ indicates that the platform can transition from $\sigma_i$ to $\sigma_j$ using the operational semantics of some instruction in Figure 3.3.

While defining the operational semantics, we axiomatize memory accesses $\mathsf{load}_\mathsf{n}$ and $\mathtt{store}_\mathsf{n}$ using the theory of arrays [105], and bit-vector operations (add, xor, etc) using SMT's bit-vector theory. We assume that attested statements are produced using a quoting scheme that is unforgeable under chosen message attacks (UF-CMA); we do not model the cryptography of this scheme, and refer the reader to [93] for a formal treatment of this subject. Therefore, we only provide an abstract description of the semantics of attest, which produces a signature over the input message and stores it in memory. The operational semantics allows rand to return an arbitrary value. The exit instruction sets the CPU to non-enclave mode, and the `pc` to an address in non-enclave memory.

$$\langle \mathsf{store_n}(e_a, e_d), \sigma \rangle \Downarrow \sigma \Big[ \mathtt{vmem} \mapsto \sigma(\mathtt{vmem})[\sigma(e_a) := \sigma(e_d)] \Big]$$

$$\langle \mathsf{v} := \mathsf{load_n}(e_a), \sigma \rangle \Downarrow \sigma \Big[ \mathsf{v} \mapsto \sigma(\mathtt{vmem})[\sigma(e_a)] \Big]$$

$$\langle \mathsf{v} := e, \sigma \rangle \Downarrow \sigma \Big[ \mathsf{v} \mapsto \sigma(e) \Big]$$

$$\langle \mathtt{jmp}\ e, \sigma \rangle \Downarrow \sigma \Big[ \mathtt{pc} \mapsto \sigma(e) \Big]$$

$$\langle \mathtt{cjmp}(\phi, e_1, e_2), \sigma \rangle \Downarrow \sigma \Big[ \mathtt{pc} \mapsto \mathsf{if}\ (\sigma(\phi))\ \{\sigma(e_1)\}\ \mathsf{else}\ \{\sigma(e_2)\} \Big]$$

$$\langle \mathtt{call}\ e, \sigma \rangle \Downarrow \sigma \Big[ \mathtt{pc} \mapsto \sigma(e),\ \mathsf{rsp} \mapsto \sigma(\mathsf{rsp} - 8),\ \mathtt{vmem} \mapsto \sigma(\mathtt{vmem})[\sigma(\mathsf{rsp} - 8) := \mathsf{next}(\sigma(\mathtt{pc}))] \Big]$$

$$\langle \mathsf{ret}, \sigma \rangle \Downarrow \sigma \Big[ \mathtt{pc} \mapsto \sigma(\mathtt{vmem})[\sigma(\mathsf{rsp})],\ \mathsf{rsp} \mapsto \sigma(\mathsf{rsp} + 8) \Big]$$

$$\langle \mathsf{attest}(e_1, e_2), \sigma \rangle \Downarrow \sigma \Big[ \mathtt{vmem} \mapsto \sigma(\mathtt{vmem})[e_2 \ldots e_2 + 428 := \mathsf{sign}(\mathtt{vmem}[e_1 \ldots e_1 + 64], \mu(curr(\sigma))) \Big]$$

$$\langle \mathsf{v} := \mathsf{rand}(), \sigma \rangle \Downarrow \sigma \Big[ \mathsf{v} \mapsto * \Big]$$

*Figure 3.3*: Operational semantics of $i \in \mathit{Instr}$: $(\sigma, \sigma') \in \rightsquigarrow$ iff $\langle i, \sigma \rangle \Downarrow \sigma'$ and $\mathtt{instr}(\sigma) = i$, where $\mathtt{instr}(\sigma)$ is the instruction to be executed next when the platform is in state $\sigma$ (computed using values of vmem and pc in $\sigma$). $\sigma \Big[ \mathsf{x} \mapsto \mathsf{y} \Big]$ denotes a state that is identical to $\sigma$, except variable x evaluates to y. The memory update expression $\mathtt{vmem}[\mathsf{x} := \mathsf{y}]$ returns a new memory that is equivalent to vmem, except for index x — multibyte-sized accesses follow the processor's endianness semantics. $\mathsf{next}(e)$ is the address of the subsequent instruction in vmem after decoding the instruction starting at address $e$.

.

## 3.6 Model of Execution within Enclaves

An enclave is a state transition system that computes by performing steps. In each step, the enclave platform first identifies the instruction to execute (based on the current state of `vmem` and `pc`), and then transitions to the next state based on the operational semantics of that instruction (which we defined in Section 3.5). Our semantics of enclave execution assumes that the platform executes instructions atomically, i.e., each step of execution is produced by an instruction invoked by the enclave — we only consider single threaded enclaves in this dissertation. Furthermore, the platform also ensures determinism modulo the input $I_e(\sigma)$, i.e., the next state of the enclave is a function of the current state $E_e(\sigma)$ and input $I_e(\sigma)$ (which includes $I_e^U(\sigma)$ and $I_e^R(\sigma)$). This is not a restriction in practice as both Sanctum and SGX enclaves interact with the external world only via memory-based I/O ($I_e^U(\sigma)$), and besides the random bits ($I_e^R(\sigma)$) from the platform's entropy source, there are no other sources of non-determinism. In other words, the privileged software only affects the enclave by controlling the input $I_e^U(\sigma)$ at each step. This property is an assumption that we make while defining the enclave's execution, so we must prove it for each enclave platform. We must prove that the platform does not contain a vulnerability that grants the privileged software another mechanism to affect the enclave's execution — as we show later (in Chapter 5), this property follows from the *secure remote execution* guarantee of the enclave platform. Note that current-generation x86 processors have several privileged-mode instructions with partially-specified behavior (e.g. SMM mode instructions); however, this does not affect our model of enclave execution because: 1) these instructions are not allowed in enclave mode of execution, and 2) these instructions do not affect enclave-specific micro-architectural state, so a privileged software attacker cannot invoke them to tamper with an enclave's execution.

The enclave computes by performing steps, where each step executes an instruction implemented by the enclave platform (which we defined in Figure 3.2 and Figure 3.3). The enclave's allowed instructions include bitvector operations on registers, memory accesses, and the invocation of primitives for generating attested statements, randomness, and exiting enclave mode. While the platform also includes privileged instructions (e.g., MSR instructions in x86) which a privileged software layer can invoke, we do not have to precisely define their semantics and simply assume that they have no effect on enclave's state $E_e(\sigma)$ — again, the platform's guarantees (specified in Chapter 5) implies that such privileged instructions only affect the privileged software's state and leave the enclave's state unmodified. This is a huge source of relief for us because modern x86 processors have several hundred such instructions, and to our knowledge, no formal specification exists for the entire instruction set.

This model of execution lets us define the platform's transition relation $\rightsquigarrow$, where ($\sigma_i$, $\sigma_j$) $\in \rightsquigarrow$ indicates that the platform can transition from $\sigma_i$ to $\sigma_j$; from hereon, we write this in infix form as $\sigma_i \rightsquigarrow \sigma_j$. When the platform transitions from $\sigma_i$ to $\sigma_j$, we also have that the enclave's state is updated from $E_e(\sigma_i)$ to $E_e(\sigma_j)$. Execution alternates between enclave code and non-enclave code, which includes the host application and privileged software layers. Therefore, a state transition may occur in either enclave or non-enclave mode — let

the function $curr(\sigma)$ denote the current mode of the platform, where $curr(\sigma) = e$ iff the platform executes enclave $e$ in state $\sigma$, else $curr(\sigma) = OS$ (we write $OS$ to mean either OS or Hypervisor, without loss of generality). For any state transition $\sigma_i \rightsquigarrow \sigma_j$ on this platform, we can derive the following statements:

1. if non-enclave software is executing in $\sigma_i$ (i.e., $curr(\sigma_i) = OS$), then $E_e(\sigma_i) = E_e(\sigma_j)$.
2. if the enclave is executing in $\sigma_i$ (i.e., $curr(\sigma_i) = e$), then $E_e(\sigma_j)$ is a function of $E_e(\sigma_i)$ and $I_e(\sigma_i)$. This is because a safe enclave platform ensures that an enclave $e$ is deterministic relative to its input $I_e(\sigma)$.

State transitions of the platform are caused either via software instructions or device I/O (via memory-mapped peripherals or direct memory access function). Since devices can produce arbitrary values, the platform executes non-deterministically, and therefore we treat $\rightsquigarrow$ as a relation as opposed to a function — the sources of non-determinism for the platform include randomness from entropy sources, direct memory accesses from I/O peripherals, instructions that observe some physical phenomena such as CPU temperature, etc. Let us refer to these bits of non-determinism in a particular state as $I^P(\sigma)$, which is only available to the privileged software — the enclave's source of non-determinism is captured in $I_e(\sigma)$, which may be derived from $I^P(\sigma)$ by the privileged software. Since $I^P(\sigma)$ captures all of the platform's non-determinism, we require that the platform's transition relation $\rightsquigarrow$ be deterministic relative to $I^P(\sigma)$.

An execution trace of the platform is an unbounded-length sequence of states denoted $\pi = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$, such that $\forall i.\ \sigma_i \rightsquigarrow \sigma_{i+1}$; $\pi[j]$ refers to the $j$th element of the trace. Since the privileged software may pause and resume $e$ at any time, we define $e$'s execution to be the subsequence of states from $\pi$ where $e$ is executing. Using the *curr* function, we can filter out the steps in $\pi$ where $e$ is not executing. We write the resulting sequence as $\langle \sigma'_0, \sigma'_1, \ldots, \sigma'_m \rangle$[2] where $init_e(E_e(\sigma'_0)) \ \wedge \ \forall i.\ curr(\sigma'_i) = e$. This subsequence is the enclave's *execution trace*: $\langle (I_e(\sigma'_0), E_e(\sigma'_0), O_e(\sigma'_0)), \ldots, (I_e(\sigma'_m), E_e(\sigma'_m), O_e(\sigma'_m)) \rangle$.

The privileged software adversary is executing concurrently with the enclave, which allows it to observe and tamper the enclave's execution (as we define in Section 4.2). For the above enclave execution trace, the adversary's observation is $\langle obs_e(\sigma'_0), obs_e(\sigma'_1), \ldots, obs_e(\sigma'_m) \rangle$. Since an execution trace of $e$ only includes the steps where $e$ invokes an instruction (whereas the platform also executes privileged software), the privileged software may provide a fresh, arbitrary input $I^U_e(\sigma)$ in any state $\sigma'$ in $\langle (I_e(\sigma'_0), E_e(\sigma'_0), O_e(\sigma'_0)), \ldots, (I_e(\sigma'_m), E_e(\sigma'_m), O_e(\sigma'_m)) \rangle$.

Finally, we define the behaviors of an enclave $e$ to be the expected execution (using the formal semantics defined in this chapter) under any input sequence. When a user provisions an enclave to a platform, she expects the platform to implement the enclave's behaviors.

**Definition 1** *The behaviors of an enclave $e$, denoted $[\![e]\!]$, is the set of finite or infinite execution traces, containing an execution trace for each input sequence comprising values of*

[2] In standard functional constructs, we have that $\langle \sigma'_0, \sigma'_1, \ldots, \sigma'_m \rangle = \text{filter}(\lambda\sigma.\ curr(\sigma) = e,\ \pi)$.

*non-enclave memory and randomness at each step of execution.*

$$[\![e]\!] = \{\langle (I_e(\sigma'_0), E_e(\sigma'_0), O_e(\sigma'_0)), \ldots \rangle \mid init_e(E_e(\sigma_0))\} \tag{3.1}$$

We must account for all potential input sequences in $[\![e]\!]$ because $e$ may receive any value of input $I_e(\sigma)$ at any step. We note that $[\![e]\!]$ may contain traces of any length, and also contain prefixes of any other trace in $[\![e]\!]$, i.e., it is prefix-closed. We adopt this definition of $[\![e]\!]$ because the privileged software can deny service by destroying the enclave at any time. Due to the determinism property of enclave programs, a specific sequence of inputs $\langle I_e(\sigma'_0), I_e(\sigma'_1), \ldots, I_e(\sigma'_m)\rangle$ uniquely identifies a trace from $[\![e]\!]$ and determines the expected execution trace of $e$ under that sequence of inputs.

## 3.7 Summary

In this chapter, we developed a formal framework for reasoning about an enclave's execution in the presence of a privileged software adversary. Specifically, we defined a formal semantics of enclave execution, which includes: 1) a precise description of enclave's state, inputs, and outputs; 2) a formalization of the syntax and semantics of enclave programs; and 3) a model of computation for enclave programs. The contribution of this chapter is necessary for our goal of formally analyzing the behavior of enclave programs in the presence of a privileged adversary, and certifying confidentiality properties of enclave programs against such a powerful adversary.

# Chapter 4

# Formal Modeling of Trusted Platforms and Privileged Adversaries

In Chapter 3, we developed a formal semantics for reasoning about enclave execution, while assuming that the platform guarantees a set of security properties to the enclave e.g. isolation of enclave private state from privileged software, etc. These properties, collectively termed *secure remote execution*, are specified and verified on models of SGX and Sanctum platforms in Chapter 5. However, any verification of an enclave platform necessitates the development of a formal model of that platform, consisting of formalization of its state and allowed operations. The focus of this chapter is to build a formal model of enclave platforms, including the first formal models of SGX and Sanctum platforms.

First, to generalize our verification of enclave platforms beyond the specifics of SGX or Sanctum, we develop an idealized abstraction, named Trusted Abstract Platform (TAP), consisting of a small set of formally-specified primitives sufficient to implement enclave execution. The TAP is a general framework for reasoning about and comparing different enclave platforms, adversary models and security guarantees. For enclave platform implementers, the TAP serves as a golden model or specification of platform behavior. From the perspective of enclave program developers, the TAP provides a means of reasoning about program security without being bogged down by implementation details of individual enclave platforms.

Next, we formalize the attacker model by defining the set of operations and observations that a privileged software adversary can perform on a TAP. This formalization is necessary for proving that an enclave platform, such as SGX and Sanctum, provide security guarantees to the enclave program.

Since we intend to use the TAP to reason about the security of Intel SGX and Sanctum, we then develop formal models of SGX and Sanctum and present machine-checked proofs showing that SGX and Sanctum are *refinements* of our idealized TAP: every operation on SGX and Sanctum, including the operations of an adversary, can be mapped to a corresponding TAP operation. Since all executions of an enclave on SGX and Sanctum can be simulated by a TAP enclave, any safety property of TAP is also a safety property of SGX and Sanctum models. There is a caveat that SGX only refines a version of TAP which leaks

some side channel observations to the attacker (see Section 4.4), therefore providing a weaker confidentiality guarantee — this form of parameterization demonstrates that the TAP allows us to develop a taxonomy of enclave platforms, each of which provides varying guarantees against different threat models.

This chapter is structured as follows. Section 4.1 defines the TAP, including its state and allowed operations. Section 4.2 formalizes the attacker's operations and observations. Section 4.3 discusses the methodology for proving that a concrete enclave platform is a refinement of the TAP, and we prove the refinement property for the SGX model in Section 4.4 and the Sanctum model in Section 4.5.

## 4.1 The Trusted Abstract Platform

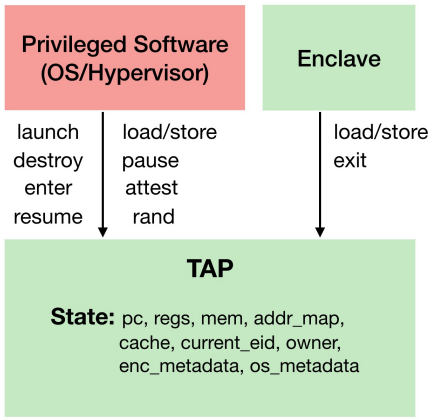



*Figure 4.1*: TAP: state and operations

The TAP exposes a set of operations to the software layers, as illustrated in Figure 4.1. It implements operations for the OS to create and schedule enclaves, and operations for the enclave program to access memory and transfer control.

The trusted abstract platform (TAP) consists of a processor with a program counter, general purpose registers, virtual address translation and a set of primitives to support enclave execution. TAP is modeled as a finite state transition system, with $\rightsquigarrow$ being the transition relation. We now define the state variables and operations supported by TAP.

### 4.1.1 TAP State Variables

The states $\Sigma$ of the TAP are a defined as a valuation of the state variables *Vars*. These variables are described in Table 4.1. `pc`, `regs`, `mem` have their usual meanings. `addr_map` maps individual virtual addresses to physical addresses and permissions. This is unlike a typical processor which uses a page table to map virtual *page numbers* to physical *page numbers*. The TAP is an abstraction and must abstract a diverse set of architectures with

| State Var. | Type | Description |
|---|---|---|
| `pc` | VA | The program counter. |
| `regs` | $\mathbb{N} \to$ W | Architectural registers: map from natural numbers to words. |
| `mem` | PA $\to$ W | Physical memory: a map from physical addresses to words. |
| `addr_map` | VA $\to$ (ACL $\times$ PA) | Map from virtual addresses to permissions and physical addresses for current process. |
| `cache` | Set $\to$ ($\mathbb{B} \times$ Tag) | Direct-mapped cache: map from cache sets to valid bits and cache tags. |
| `current_eid` | $\mathcal{E}_\mathcal{I}$ | Current enclave. `current_eid` $= \mathcal{OS}$ means that no enclave is being executed. |
| `owner` | PA $\to \mathcal{E}_\mathcal{I}$ | Maps a physical address to the enclave (or OS) that has access rights to that address. |
| `enc_metadata` | $\mathcal{E}_\mathcal{I} \to \mathcal{E}_\mathcal{M}$ | Map from enclave ids to metadata record type ($\mathcal{E}_\mathcal{M}$). |
| `os_metadata` | $\mathcal{E}_\mathcal{M}$ | Record that stores a checkpoint of privileged software state. |

*Table 4.1*: Description of TAP State Variables.

different page table structures and page sizes. Therefore, it maps each virtual address to a physical address.

The TAP includes a model of `cache` which is used to show that the TAP preserves confidentiality in the presence of a software adversary attempting cache attacks. The cache model is of a physically-indexed physically-tagged direct-mapped cache. Note that ensuring confidentiality for a direct-mapped cache also ensures confidentiality for set-associative caches. The TAP cache model leaves the mapping of physical addresses to cache sets and mapping of physical addresses to cache tags uninterpreted. In other words, the TAP formalism only requires that the cache set and cache tag for each memory access be deterministic functions of the physical address. The exact function is specified by implementations (refinements) of TAP, which are models of SGX and Sanctum in this paper.

The variable `current_eid` tracks the enclave currently being executed, and it equals $\mathcal{OS}$ if the CPU is in non-enclave mode . The variable `owner` maps each physical address to the enclave which exclusively "owns" it. If `owner`$[p] = e$, only enclave $e$ can access (fetch/read/write) this word of memory. We abuse notation and use $e$ to refer to both the "enclave id," a unique integer assigned by the platform to an enclave, as well the enclave itself. Attempts to access physical address $p$ by all other enclaves and privileged software are blocked. `owner`$[p] = \mathcal{OS}$ means that address $p$ is not allocated to any enclave. `owner` corresponds to the EPCM in SGX and the DRAM bitmap in Sanctum. It is the primary mechanism to enforce isolation of enclave's private memory. `enc_metadata` stores metadata

about each initialized enclave.

| State var. | Description |
|---|---|
| `entrypoint` | Enclave entrypoint. |
| `addr_map` | Virtual to physical mapping/permissions. |
| `excl_vaddr` | Set of private virtual addresses. |
| `measurement` | Enclave measurement. |
| `pc` | Saved PC (in case of interrupt). |
| `regs` | Saved registers (in case of interrupt). |
| `paused` | Flag set only when enclave is interrupted. |

*Table 4.2*: Fields of the `enc_metadata` record.

*Enclave Metadata*: Table 4.2 lists various fields within the enclave metadata record. It stores the entrypoint to the enclave, its virtual to physical mappings and what set of virtual addresses are private to the enclave. The `pc` and `regs` fields are used to checkpoint enclave state when it is interrupted. The `paused` flag is set to `true` only when an enclave is interrupted and ensures that enclaves cannot be tricked into resuming execution from invalid state.

*Privileged Software Metadata*: The `os_metadata` record contains three fields: `pc`, `regs`, and `addr_map`. The `pc` and `regs` fields store a checkpoint of privileged software state. These are initialized when entering enclave state and restored when the enclave exits. The `addr_map` field is the privileged software's virtual to physical address mapping and associated permissions.

### 4.1.2 TAP Operations

Table 4.3 describes the operations supported by the TAP. We present the detailed specification (in the form of reference implementation) of these operations in Appendix A, and use this section to present a general overview of their semantics. The operations `fetch`, `load`, `store` work as usual. The platform guarantees that memory owned by enclave $e$ is not accessible to other enclaves or privileged software. Each of these operations update cache state, set the access bit in `addr_map`, and return whether the operation was a hit or a miss in the cache.

The virtual to physical mappings of both enclave and privileged software are controlled using `get_addr_map` and `set_addr_map`. As enclave's memory access pattern can leak via observation of the access/present bits in `addr_map`. `get_addr_map`$(e, v)$ must fail (on a secure TAP) for virtual addresses in the set `enc_metadata`$[e]$`.evrange` when called from outside the enclave. However, SGX does permit privileged software to access an enclave's private page tables, which leads to a side channel leak. Therefore, in order to model platforms

| **Operation** | **Description** |
|---|---|
| `fetch`($v$)<br>`load`($v$)<br>`store`($v$) | Fetch/read/write from/to virtual address $v$. Fail if $v$ is not executable/readable/writeable respectively according to the `addr_map` or if `owner`[`addr_map`[$v$].PA] $\neq$ `current_eid`. |
| `get_addr_map`($e, v$)<br>`set_addr_map`($e, v, p, perm$) | Get/set virtual to physical mapping and associated permissions for virtual address $v$. |
| `launch`($e, m, x_v, x_p, t$) | Initialize enclave $e$ by allocating `enc_metadata`[$e$]. |
| `destroy`($e$) | Set `mem`[$p$] to 0 for each $p$ such that `owner`[$p$] = $e$. Deallocate enclave `enc_metadata`[$e$]. |
| `enter`($e$), `resume`($e$) | `enter` enters enclave $e$ at entrypoint, while `resume` starts execution of $e$ from the last saved checkpoint. |
| `exit`(), `pause`() | Exit enclave. `pause` also saves a checkpoint of `pc` and `regs` and sets `enc_metadata`[$e$].`paused` = `true`. |
| `attest`($d$) | Return hardware-signed message containing data $d$ and $e$'s measurement: $d \mid\mid \mu(e) \mid\mid \{d \mid\mid \mu(e)\}_{SK_p}$. |
| `rand` | Return a random byte from a cryptographically-secure source of entropy |

*Table 4.3*: Description of TAP Operations

$$
\begin{aligned}
&\big(\texttt{current\_eid} = \mathcal{OS} \wedge e \notin \texttt{enc\_metadata} \wedge \\
&\ \mathit{executable}(m[t]) \wedge t \in x_v \wedge \\
&\ \forall p.\ p \in x_p \implies \texttt{owner}[p] = \mathcal{OS} \wedge \\
&\ \forall v.\ v \in x_v \implies (\mathit{valid}(m[v]) \implies m[v]_{\mathrm{PA}} \in x_p) \wedge \\
&\ \forall v_1, v_2.\ (v_1 \in x_v \wedge v_2 \in x_v) \implies (m[v_1]_{\mathrm{PA}} \neq m[v_2]_{\mathrm{PA}})\big) \\
&\quad \iff (\texttt{launch\_status} = \texttt{success})
\end{aligned}
$$

*Figure 4.2*: Conditions for the success of `launch`. Note that $m[v]_{\mathrm{PA}}$ refers to physical address that virtual address $v$ points to under the mapping $m$.

such as SGX, we introduce a "setting" in the TAP, called `priv_mappings`, and this insecure behavior is allowed when `priv_mappings` = `false`.

*Enclave Creation*: The `launch`$(e, m, x_v, x_p, t)$ operation is used to create an enclave. The enclave's virtual to physical address mapping and associated permissions are specified by $m$. $x_v$ is the set of enclave-private *virtual* addresses (*evrange*). It corresponds to the base address and size arguments passed to `ecreate` in SGX and `create_enclave` in Sanctum. $x_p$ is the set of *physical* addresses allocated to the enclave and its entrypoint is the virtual address $t$. The `launch` operation only succeeds if enclave $e$ does not already exist, if the entrypoint is mapped to an enclave-private executable address, every virtual address in $x_v$ that is accessible to an enclave points to a physical address in $x_p$, and if there is no aliasing among the addresses in $x_v$. A precise statement of the conditions that result in a successful launch shown in Figure 4.2. These conditions have subtle interactions with the requirements for SRE. For example, if virtual addresses within $x_v$ are allowed to alias, an adversary can construct two enclaves which have the same measurement but different semantics. The potential for such attacks emphasizes the need for formal modeling and verification.

*Enclave Destruction*: An enclave is deallocated when the OS invokes `destroy`, which zeroes out the enclave's memory so that its private state is not leaked when the privileged software reclaims the memory. This is necessary for confidentiality because untrusted privileged software can destroy an enclave at any time. This operation fails if the enclave is currently running — the OS must either invoke pause or the enclave must have exited prior to its destruction.

*Enclave Entry/Exit*: the untrusted OS invokes `enter`$(e)$ to transfer control to the enclave $e$ at its configured entry point. The operation `enter`$(e)$ enters enclave $e$ by setting the `pc` to its $\mathit{config}_e.\mathit{entrypoint}$ and `current_eid` to $e$. The enclave may transfer control back to the calling OS via `exit`.

*Enclave Pause/Resume*: `pause` is invoked by the untrusted OS to pause execution with enclave $e$ (typically via a device interrupt), and return control back to the OS — TAP ensures that the instructions are atomic, so execution stops at the instruction boundary and the TAP

saves the current state of the enclave (general purpose registers, flags, etc.) within $e$'s private memory. `resume`($e$) is invoked by the untrusted OS to transfer control to the enclave $e$. The TAP reloads the enclave's state at the time of the last pause. Note that invoking resume only succeeds if the enclave was pause-ed (as opposed to the enclave performing an exit).

*Attested Statements*: The attestation operator provided by the TAP ensures that the user is communicating with a bona fide enclave. The `attest` operation can only be invoked from within the enclave and may be used by the enclave to establish its identity as part of an authentication protocol. `attest` returns a hardware-signed cryptographic digest of data $d$ and a measurement: $d \mid\mid \mu(e) \mid\mid \{d \mid\mid \mu(e)\}_{SK_p}$. The signature uses the processor's secret key $SK_p$, whose corresponding public key is signed by the trusted platform manufacturer.

*Random Number Generation*: The `rand` primitive is invoked by the enclave code to get a cryptographically secure random byte. Enclaves typically use this for cryptography e.g. randomized encryption.

### 4.1.3 Enclave State, Inputs, and Outputs and the Adversary's State

We now precisely define the enclave's state, inputs, and outputs. Recall from Section 3.6 that the state of an enclave $e$ is denoted by $E_e(\sigma)$. $E_e(\sigma)$ is defined as the tuple $\langle E_{\texttt{vmem}}(e,\sigma), E_{\texttt{regs}}(e,\sigma), E_{\texttt{pc}}(e,\sigma), E_{\texttt{cfg}}(e,\sigma)\rangle$ if $e \in$ `enc_metadata` and $\perp$ otherwise. The components of this tuple are as follows:

- $E_{\texttt{vmem}}(e,\sigma)$ is a partial map from virtual addresses to words. It is defined to be $\sigma(\texttt{mem}[\texttt{enc\_metadata}[e].\texttt{addr\_map}[v]_{\text{PA}}])$ if $v \in \sigma(\texttt{enc\_metadata}[e].\texttt{evrange})$ and $\perp$ otherwise. In other words, $E_{\texttt{vmem}}$ refers to the content of each virtual memory address in the enclave's `evrange`.

- $E_{\texttt{regs}}(e,\sigma)$ denotes the registers. It is $\sigma(\texttt{regs})$ if $curr(\sigma) = e$ (when the enclave is executing), and $\sigma(\texttt{enc\_metadata}[e].\texttt{regs})$ otherwise.

- $E_{\texttt{pc}}(e,\sigma)$ denotes the program counter. It is $\sigma(\texttt{pc})$ if $curr(\sigma) = e$ (when the enclave is executing), and $\sigma(\texttt{enc\_metadata}[e].\texttt{pc})$ otherwise.

- The tuple $E_{\texttt{cfg}}(e,\sigma)$ consists of the following elements:

  (i) $\sigma(\texttt{enc\_metadata}[e].\texttt{addr\_map})$
  (ii) $\sigma(\texttt{enc\_metadata}[e].\texttt{entrypoint})$
  (iii) $\sigma(\texttt{enc\_metadata}[e].\texttt{evrange})$

Recall from Section 3.6 that the input to enclave $e$ at state $\sigma$ is denoted by $I_e(\sigma)$, where $I_e(\sigma) \doteq \langle I_e^R(\sigma), I_e^U(\sigma)\rangle$. $I_e^R(\sigma)$ is the random number provided at the state $\sigma$. $I_e^U(\sigma)$ refers to the contents of each virtual address *not* in the enclave's `evrange`. Formally, $I_e^U(\sigma)$ is a partial map from virtual address to words. It is $\sigma(\texttt{mem}[\texttt{enc\_metadata}[e].\texttt{addr\_map}[v]_{\text{PA}}])$ if each of these conditions hold: (i) enclave $e$ is executing: $curr(\sigma) = e$, (ii) $v$ is mapped to

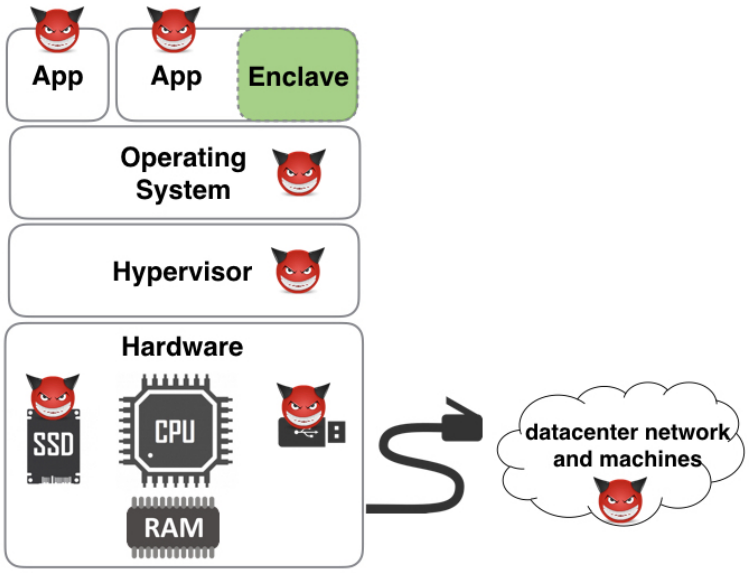



*Figure 4.3*: Threat Model: The adversary controls the OS, hypervisor, all hardware except the CPU and RAM, and other machines on the network. The enclave is the only trusted software component.

some physical address: $\sigma(valid(\texttt{enc\_metadata.addr\_map}[v]))$, and (iii) $v$ is not private: $v \notin \sigma(\texttt{enc\_metadata}[e].\texttt{evrange})$; it is $\perp$ otherwise.

The output of enclave $e$ in state $\sigma$ is denoted by $O_e(\sigma)$. Since the output $O_e(\sigma)$ contains memory values outside enclave's `evrange`, it is defined identically to $I_e^U(\sigma)$.

Finally, we formalize the TAP adversary's state. Since the privileged software adversary can access any component of the platform's state outside an enclave's protected memory, we define the adversary's state with respect to an enclave $e$. From the perspective of an enclave $e$, the adversary's state $A_e(\sigma)$ is modeled as the tuple $\langle S(\sigma), \hat{E}_e(\sigma)\rangle$. $S(\sigma) \doteq \langle S_{\texttt{vmem}}(\sigma), S_{\texttt{regs}}(\sigma), S_{\texttt{cfg}}(\sigma)\rangle$. $S_{\texttt{vmem}}(\sigma) \doteq \lambda v.\ \sigma(\texttt{mem}[\texttt{os\_metadata.addr\_map}[v]_{\text{PA}}])$ is its view of memory. $S_{\texttt{regs}}(\sigma)$ denotes the privileged software's registers: $\sigma(\texttt{regs})$ when the adversary is executing, and $\sigma(\texttt{os\_metadata.regs})$ otherwise. $S_{\texttt{cfg}}(\sigma)$ is the privileged software's page mapping: $\sigma(\texttt{os\_metadata.addr\_map})$. $\hat{E}_e(\sigma)$ is the state of all the other enclaves in the system except for enclave $e$: $\hat{E}_e(\sigma) \doteq \lambda e'.\ ITE(e \neq e', E_{e'}(\sigma), \perp)$, where $ITE$ abbreviates if-then-else expression.

## 4.2 Formal Model of a Privileged Adversary

Security properties of the enclave program must be stated relative to a threat model, which we formalize in this section. The illustration of our threat model in Figure 8.1 lists all the system components that are under the attacker's control — we only model privileged software-level attacks in this work. The adversary may also control all of the system software in the host computer, including the operating system and the hypervisor. The adversary may

fully control all of the peripherals in the host computer, including disks and network cards. This allows the adversary to record, replay, and modify all I/O communication, including network packets and files, since all I/O is proxied through unprotected memory. We assume that the attacker cannot physically extract secrets from the processor and DRAM — that being said, SGX further removes the DRAM chip from the trusted computing base using memory encryption and integrity verification, but other enclave platforms such as Sanctum do not defend against attacks on DRAM chips. This adversary is general enough to model privileged malware running in the OS and hypervisor layers, as well as a malicious system administrator who may try to access the data by logging into the host and inspecting disks and memory. Denial-of-service attacks are out of scope — the cloud provider or attacker may choose to not run the enclave program or delete all of its data.

In this section, we define the effect of the adversary's operations and observations of an enclave's execution. Instead of defining the adversary's operations and observations separately for each enclave platform, we define them for the TAP. Since each operation on the refined model (e.g. Sanctum) can be simulated by the TAP, and since the privileged software attacker can invoke any operation on the platform, it is sound to model the adversary's actions at the level of TAP i.e. we do not restrict a Sanctum attacker's abilities by specifying them at the level of TAP.

### 4.2.1 Operations of a TAP Adversary

An enclave executes in the presence of such a privileged software adversary, and an enclave platform is not required to fully isolate the enclave from the adversary; in fact, recall that the attacker may access the inputs and outputs of the enclave, which is acceptable because the I/O over the network should be considered untrusted. In this threat model, the attacker may force the CPU to transfer control from the enclave to the privileged malware at any time; for instance, the adversary may configure devices to generate interrupts, which is handled by the OS/VMM. Once the CPU transfers out of the enclave, the adversary may execute an arbitrary set of instructions before transferring control back to the enclave. In other words, an adversary performs an unbounded number of adversarial operations between any pair of instructions executed by enclave. The most general adversary, termed the $MCP$ adversary, is allowed to perform an unbounded number of the following actions:

1. Unconstrained updates to `pc` and `regs`.
2. Loads and stores to memory with arbitrary address ($va$) and data ($data$) arguments.
   - $\langle op, hit_f \rangle \leftarrow \texttt{fetch}(va)$
   - $\langle \texttt{regs}[ri], hit_l \rangle \leftarrow \texttt{load}(va)$
   - $hit_s \leftarrow \texttt{store}(va, data)$
3. Modification of the adversary's page-tables by calling `set_addr_map` and `get_addr_map` with unconstrained arguments.

- `set_addr_map`$(e, v, p, perm)$
- `regs`$[ri] \leftarrow$ `get_addr_map`$(e, v)$

4. Invocation of TAP's enclave operations with unconstrained arguments.
   - Launch enclaves with arbitrary code: `launch`$(e, m, x_v, x_p, t)$
   - Destroy any enclave: `destroy`$(e)$
   - Enter and resume any enclaves: `enter`$(e)$ and `resume`$(e)$
   - Exit (`exit`) from and interrupt (`pause`) any enclave

Any adversarial computation, including malicious operating systems, hypervisors, host applications, and even other malicious enclaves, can be modeled using a combination of the aforementioned actions. By proving TAP's security guarantees against such an threat model allows us to claim that TAP provides SRE guarantee against a general, privileged software adversary, which we call the the $MCP$ adversary.

*Restricted Adversaries*: In addition to $MCP$, we define restricted adversaries $MC$ and $M$, and use them to prove SRE guarantee of SGX against a weaker attacker with limited observations. The $MC$ adversary is restricted to computation based on memory values and cache state; it ignores the value returned by `get_addr_map`. The $M$ adversary only computes using memory values; it ignores $hit_f, hit_l$ and $hit_m$ returned by `fetch`, `load` and `store`, respectively, in addition to the result of `get_addr_map`.

### 4.2.2 Observations of a TAP Adversary

The observation function captures what state the user expects to be attacker-visible.
**Adversary** $M$: The baseline adversary $M$ only observes the outputs produced by an enclave i.e. any write by the enclave to non-enclave memory. The observation function $obs_e^M(\sigma)$ is a partial map from physical addresses to words and allows the adversary to observe the contents of all memory locations not private to enclave $e$. It is equal to the enclave's output in state $\sigma$ — more precisely, it is defined as $\sigma(\texttt{mem}[p])$ when $\sigma(\texttt{owner}[p]) \neq e$ and $\perp$ otherwise.

$$obs_e^M(\sigma) \doteq O_e(\sigma) \doteq \lambda p.\ ITE(\sigma(\texttt{owner}[p]) \neq e, \sigma(\texttt{mem}[p]), \perp)$$

**Adversary** $MC$: The observation function $obs_e^{MC}(\sigma)$ specifies that besides contents of memory locations that are not private to an enclave, the adversary can also observe whether these locations are cached. It is also a partial map from physical addresses to words and is defined to be the tuple $\langle \sigma(\texttt{mem}[p]), cached(\sigma, p) \rangle$ when $\sigma(\texttt{owner}[p]) \neq e$ and $\perp$ otherwise. $cached(\sigma, p)$ is true iff physical address $p$ stored in the cache in the machine state $\sigma$.

$$obs_e^{MC}(\sigma) \doteq \lambda p.\ ITE(\sigma(\texttt{owner}[p]) \neq e, \langle \sigma(\texttt{mem}[p]), cached(\sigma, p) \rangle, \perp)$$

Note that the adversary cannot directly observe whether an enclave's private memory locations are cached. However, unless cache sets are partitioned between the attacker and the enclave, cache attacks [129, 120] allow the adversary to *learn* this information.

**Adversary** $MP$: The observation function $obs_e^{MP}(\sigma)$ specifies that besides contents of memory locations that are not private to an enclave (i.e. $M$'s observation function), the adversary can also observe the virtual to physical mappings and associated access / permission bits for each virtual address.

$$obs_e^{MP}(\sigma) \doteq \langle obs_e^{M}(\sigma), \lambda v.\ \sigma(\texttt{get\_addr\_map}(e, v)\rangle$$

The notation $\sigma(\texttt{get\_addr\_map}(e, v))$ refers to the result of evaluating $\texttt{get\_addr\_map}(e, v)$ in the state $\sigma$.
**Adversary** $MCP$: This observation function includes the observations for all aforementioned adversaries. Specifically, $obs_e^{MCP}(\sigma)$ includes observation of memory that is not private to the enclave $e$, presence or absence of each virtual address (within enclave's private memory) in the cache, virtual to physical mappings and associated access/permission bits for each virtual address (within enclave's private memory).

$$obs_e^{MP}(\sigma) \doteq \langle obs_e^{MC}(\sigma), obs_e^{MP}(\sigma)\rangle$$

## 4.3 Refinements of the TAP

We prove that models of MIT Sanctum and Intel SGX are *refinements* of the TAP under certain adversarial parameters. Refinement shows that each operation, including all adversarial operations, on Sanctum and SGX processors can be mapped to an "equivalent" TAP action. The refinement proof implies SRE, which was proven on the TAP, also holds for for our models of SGX and Sanctum.

### 4.3.1 Refinement Methodology

Let $Impl = \langle \Sigma_I, \rightsquigarrow_I, init_I\rangle$ be a transition system with states $\Sigma_I$, transition relation $\rightsquigarrow_I$ and initial state $init_I$. We say that $Impl$ refines $TAP$, or equivalently $TAP$ simulates $Impl$, if there exists a *simulation relation* $R \subseteq (\Sigma_I \times \Sigma)$ with the following property:

$$\Big(\forall s_j \in \Sigma_I, s_k \in \Sigma_I, \sigma_j \in \Sigma. \tag{4.1}$$
$$(s_j, \sigma_j) \in R \wedge s_j \rightsquigarrow_I s_k \implies$$
$$\big((s_k, \sigma_j) \in R \vee (\exists \sigma_k \in \Sigma.\ \sigma_j \rightsquigarrow \sigma_k\ \wedge\ (s_k, \sigma_k) \in R)\big)\Big) \wedge$$
$$(init, init_I) \in R$$

The condition states that for every pair of states $(s_j, \sigma_j)$ (belonging to $Impl$ and $TAP$ respectively) that are related by $R$, if $Impl$ steps from $s_j$ to $s_k$, then either (i) the $TAP$ takes

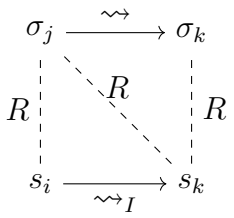



*Figure 4.4*: Illustration of Stuttering Simulation.

no steps, and $s_k$ is related to $\sigma_j$, or (ii) there exists a state $\sigma_k$ of *TAP* such that $\sigma_j$ steps to $\sigma_k$ and $(s_k, \sigma_k)$ are related according to $R$. In addition, the initial states of *Impl* and *TAP* must be related by $R$. This is illustrated in Figure 4.4. This corresponds to the notion of stuttering simulation [30], and we require stuttering because a single invocation of `launch` corresponds to several API calls in Sanctum and SGX.

Refinement states that every trace of *Impl* can be mapped using the relation $R$ to a trace of *TAP*; effectively this means that *TAP* has a superset of the behaviors of *Impl*. The security properties of the TAP are *hyperproperties*, which in general, are not preserved by refinement [36]. However, the properties we consider are *2-safety* properties [118] that are variants of *observational determinism* [81, 98]. These are properties *are preserved* by refinement. Therefore, the SRE properties proven on the TAP also hold for SGX and Sanctum, as we show that these architectures (conditionally) refine the TAP.

## 4.4 Refinement of the TAP: Intel SGX

Intel Software Guard Extensions extend the Intel architecture to enable execution of enclave programs.

### 4.4.1 SGX Overview

Enclaves in SGX are implemented in hardware (microcode), which provides an instruction set extension [61] to create enclaves (`ecreate`), enter enclaves (`eenter`), generate attested statements (`ereport`), etc. The SGX processor dedicates a contiguous region of physical memory (called the enclave page cache, or EPC), exclusively useable for storing enclave pages. While this provides confidentiality of enclave's memory, SGX does not protect several side channel leaks such as cache timing or page access patterns. The last level cache is shared between the enclave and the adversary, and adversary memory can map to the same cache set as the enclave's private address, allowing the attacker to perform cache attacks [120, 129, 82, 28]. SGX also allows the OS to examine the page tables that control the enclave's private memory, enabling the OS to read the accessed and dirty bits, thus learning the enclave's memory access pattern at the page-level granularity. The OS also gets notified on page fault

exceptions (as part of demand paging), and this is another channel to learn the enclave's page-level memory access patterns [109, 125].

### 4.4.2 SGX Model

We create a model of the SGX platform at the level of abstraction presented in the Intel Programmer Reference manual [61]. The model contains ISA-level semantics of the SGX instructions. However, for simplicity, it elides details such as attestation, hardware encryption of DRAM pages, and the cryptographic protections (encryption, integrity, and freshness) of demand paging and instead assumes several axioms about these features.

```
1 procedure ecreate(mem: [bv64] bv8, rbx: bv64, rcx: bv64)
2 {
3   var srcpge: bv64;
4   var secs: bv64;
5   var measurement: bv256;
6
7   assert ...; //a set of security checks
8
9   srcpge := pageinfo_srcpge(mem, rbx); //source SECS page
10  secs := rcx; //rcx holds the destination SECS page
11  mem[secs..secs+4096] := mem[srcpge..srcpge+4096]; //memcpy SECS page
12
13  epcm[secs] := create_epcm(true, PT_SECS, 0, 0);
14  measurement := SHA256(SECS_MRENCLAVE(secs_base) || SECS_SIZE(secs_base) || ...);
15  mem[SECS_MEASUREMENT_OFFSET(secs)..SECS_MEASUREMENT_OFFSET(secs) + 256] := measurement;
16 }
17
18 procedure load_8(mem: [bv64] bv8, va: bv64) : bv8
19 {
20  var check : bool; //EPCM security checks succeed?
21  var pa : bv64; //translated physical address
22  var ea : bool; //enclave access to enclave memory?
23  pa := pagewalk(mem, va);
24  ea := CR_ENCLAVE_MODE && evrange(va);
25  check := epc(pa) &&
26           EPCM_VALID(epcm[pa]) &&
27           EPCM_PT(epcm[pa]) == PT_REG &&
28           EPCM_ENCLAVESECS(epcm[pa]) == CR_ACTIVE_SECS &&
29           EPCM_ENCLAVEADDRESS(epcm[pa]) == va;
30  assert (ea => check); //EPCM security checks
31  assert ...; //read bit set and pagetable has valid mapping
32  if (!ea && epc(pa)) {return 0xff;} else {return mem[pa];}
33 }
34
35 procedure eexit(mem: [bv64] bv8, rbx: bv64)
36 {
37  regs[rip] := rbx;
38  regs[CR_ENCLAVE_MODE] := false;
39  mem[CR_TCS_PA] := 0x00;
40 }
```

*Figure 4.5*: Models for `ecreate`, `load`, and `eexit` instructions

The machine's state is a valuation of several state variables: `mem`, `regs`, and `epcm`. As

their names suggest, mem denotes physical memory, regs is a collection of ISA-visible CPU registers (e.g. rax, CR4), and epcm denotes the enclave metadata maintained by the CPU (stored in CPU hardware) for various security checks. Physical memory mem is modeled as a flat array, with index type of 64 bits and element type of 8 bits. mem is partitioned by the platform into two disjoint regions: protected memory ($\texttt{mem}_{\texttt{epc}}$)[1] for use by all enclaves on the machine, and unprotected memory ($\texttt{mem}_{\neg\texttt{epc}}$) for use by all other software on the machine. Correspondingly, we define a predicate epc such that for any physical address a, epc(a) is true iff a is an address in $\texttt{mem}_{\texttt{epc}}$. Furthermore, a particular enclave program owns a protected, contiguous region of virtual memory, called evrange, whose pages are mapped to a subset of physical pages within $\texttt{mem}_{\texttt{epc}}$. For any virtual address a, evrange(a) is true iff a is within evrange. The epcm is a finite sized array of hardware-managed structures, where each structure stores security-critical metadata about a page in $\texttt{mem}_{\texttt{epc}}$ (e.g. page permissions, owner enclave, etc.).

Each SGX instruction is modeled as a stateful computation that takes a set operands as inputs, produces an output, and updates the machine state. Figure 4.5 presents formal models of sample SGX instructions ecreate (for launching new enclaves) and eexit (for transferring control from an enclave to untrusted code), and the load instruction to illustrate the revised memory access control in x86-64 to enable enclave memory — models of other SGX instructions are included in Appendix B. These models are manually derived from the Intel SGX reference [61], and we specify them in the Boogie verification language [42]; we refer the reader to [42] for documentation on Boogie's syntax, semantics, and its verification backend based on First-Order Logic theorem proving. Each instruction is modeled as a Boogie procedure containing stateful computation that optionally produces an output (e.g. 8-bit value from load). By using Boogie's translation to First-Order Logic theories, we have effectively defined the denotational semantics of SGX instructions in a combination of First-Order theories such as Bitvectors, Arrays, and Uninterpreted Functions — for instance, load and store primitives perform array reads and writes on mem and epcm, which are interpreted by the Theory of Arrays [17], and arithmetic operations on registers are encoded in the Theory of Bitvectors [105]. We now describe the sample instructions in Figure 4.5.

**ecreate:** The ecreate instruction is used to initiate the creation of new enclaves — the enclave is not valid until the OS issues an einit instruction, which marks the enclave valid and ready to be entered. After performing a range of security checks (not shown in the model in Figure 4.5), the ecreate procedure allocates a page to hold a structure, called the SECS, which is used throughout the duration of the enclave's lifecycle for various security related checks. The SECS structure stores security-critical metadata such as the enclave's measurement, the base and high addresses of its evrange, etc. The enclave's measurement is partially computed in ecreate (by hashing various attributes such as the evrange), and will be updated in future calls to eextend and einit.

[1] As per Intel's terminology, epc is an abbreviation of enclave page cache

`load`: The `load` procedure models the memory access control implemented in Intel SGX [80]; we model both traditional checks (e.g. permission bits, valid page table mapping, etc.) and SGX-specific security checks. First, `load` reads the page table to translate the virtual address `va` to physical address `pa` (line 23) using a traditional page walk. The boolean flag `ea` denotes whether this access is made by enclave code to an address within evrange. If `ea` is true, then `load` asserts (line 30) that the following security checks succeed:

- the translated physical address `pa` resides in $\texttt{mem}_{\texttt{epc}}$ (line 25)
- `epcm` array contains a valid entry for address `pa` (lines 26 and 27)
- the `epcm` entry (for address `pa`) belongs to the currently running enclave (line 28)
- the address mapping and permission bits in the page table are same as when enclave was initialized (line 29)

If non-enclave code is accessing $\texttt{mem}_{\texttt{epc}}$, or if the enclave is attempting to access some other enclave's memory (i.e. within $\texttt{mem}_{\texttt{epc}}$ but outside its evrange), then `load` returns a dummy value `0xff` (line 32); if any security check fails, then a page fault exception is raised (lines 30 and 31). We refer the reader to [80] for details on SGX memory access semantics.

`eexit`: Invoking `eexit` causes the control flow to transfer to the host application. As seen in Figure 4.5, while evaluating `eexit`, the SGX processor is set to non-enclave mode, and the program counter is set to an address in non-enclave memory (outside evrange).

### 4.4.3 SGX Model Refines TAP

We attempted to prove that SGX refines TAP, in that all SGX traces can be mapped to TAP traces. However, we cannot prove SGX refinement unconditionally. We show that refinement holds only when `priv_mappings` = `false` (see Sec. 4.1.2). This is because SGX implements a mechanism for the attacker (OS) to view page table entries, which contains the accessed and dirty bits. As TAP confidentiality for Adversary $MCP$ only holds when `priv_mappings` = `true`, SGX is not secure against $MCP$. Furthermore, the lack of cache partitioning also prevents us from showing that Equation 5.5 holds, so SGX does not refine TAP instantiated with Adversary $MC$. We *are* able to prove refinement of TAP by SGX for the restricted adversary $M$. This shows SGX provides similar guarantees to Sanctum, except for leakage through the cache and page table side-channels.

## 4.5 Refinement of the TAP: Sanctum Processor

Sanctum [38] is an open-source enclave platform that provide strong confidentiality guarantees against privileged software attackers. In addition to protecting enclave memory from direct observation and tampering, Sanctum protects against software attackers that seek to observe an enclave's memory access patterns.

### 4.5.1 Sanctum Overview

Sanctum implements enclaves via a combination of hardware extensions to RISC-V [11] and trusted software at the highest privilege level, called the *security monitor*.

**Sanctum Hardware**: Sanctum minimally extends Rocket Chip [11, 10], an open source reference implementation of the RISC-V ISA [124, 123]. Specifically, Sanctum hardware isolates physical addresses by dividing system memory (DRAM) into *regions*, which use disjoint last level cache sets, and allocating each region to an enclave exclusively. Since enclaves have exclusive control over one or more DRAM regions, there is no leakage of private memory access patterns through the cache. An adversary cannot create a TLB entry that maps its virtual address to an enclave's private cache set.

**Sanctum Monitor**: The bulk of Sanctum's logic is implemented in a trusted *security monitor*. The monitor exclusively operates in RISC-V's *machine mode*, the highest privilege-level implemented by the processor, and solely able to bypass virtual address translation. Monitor data structures maintain enclave and DRAM region state. The monitor configures the Sanctum hardware to enforce low-level invariants that comprise enclave access control policies. For example, the monitor places an enclave's page tables within that enclave's DRAM region, preventing the OS from monitoring an enclave's page table metadata to infer memory access patterns. The monitor exposes an API for enclave operations, including measurement. A trusted bootloader bootstraps the system, loads the monitor and creates a chain of certificates authenticating the Sanctum chip and the loaded security monitor.

### 4.5.2 Sanctum Model

Our Sanctum model combines the Sanctum hardware and the reference security monitor, and includes hardware registers, hardware operations, monitor data structures and the monitor API. The hardware registers include the DRAM bitmap which tracks ownership of DRAM regions, page table base pointers, and special regions of memory allocated to the monitor and for direct memory access (DMA). Hardware operations modeled include page tables and address translation, and memory instruction fetch, loads and stores. The reference implementation and golden specification of the monitor (i.e., Sanctum platform's primitives) are included in Appendix C.

### 4.5.3 Sanctum Model Refines TAP

The Sanctum refinement proof is expressed in three parts.

**1. Concrete MMU refines Abstract MMU**: We constructed an abstract model of a memory management unit (MMU). The abstract MMU's address translation is similar to the TAP; it is a single-level map from virtual page numbers to permissions and physical page numbers. In contrast, the concrete Sanctum MMU models a multi-level page table walk in memory. We showed that the concrete Sanctum MMU model refines the abstract MMU. We then used the abstract MMU in modeling the Sanctum Monitor. This simplifies the simulation relation between Sanctum and TAP.

**2. Sanctum Simulates TAP**: We showed that every Sanctum state and every Sanctum operation, which includes both enclave and adversary operations, can be mapped to a corresponding TAP operation. For this, (i) we constructed a simulation relation between Sanctum states and corresponding TAP states, and (ii) we constructed a *Skolem function* [114] mapping each Sanctum operation to the corresponding TAP operation. We proved that for every pair of states in the simulation relation, every Sanctum operation can be mapped by the Skolem function to a corresponding TAP operation such that the resultant states are also in the simulation relation.
**3. Proof of Cache Partitioning**: The Sanctum model instantiates the function *pa2set* which maps physical addresses to cache sets. We showed that the Sanctum API's `init_enclave` operation and the definition of *pa2set* together ensure that all Sanctum enclaves' cache sets are partitioned, i.e., Equation 4.2 is satisfied.

$$\forall p_1, p_2, \sigma, e. \qquad (4.2)$$
$$\sigma(\texttt{owner}[p_1] = e \wedge \texttt{owner}[p_2] \neq e) \implies (pa2set(p_1) \neq pa2set(p_2))$$

## 4.6 Summary

This chapter developed formal models enclave platforms and several models of the privileged software attacker, with varying capabilities. First, we defined an abstract model of an enclave platform, called the Trusted Abstract Platform (TAP), and prove that concrete enclave platforms, such as Intel SGX and Sanctum, are refinements of the TAP model — models of SGX and Sanctum platforms are novel contributions of this dissertation. The refinement guarantee establishes that any operation on a SGX or Sanctum platform can be simulated by an operation exposed by the TAP; therefore any safety or k-safety property [36] that we prove of the TAP is also satisfied by the concrete SGX and Sanctum platforms. This is extremely valuable for us in Chapter 5 where we prove that TAP provides the secure remote execution property to all enclave programs. By defining the TAP abstraction, we are now equipped to reason about enclave programs generally, and are able to port an enclave program onto other enclave platforms (that are refinements of TAP) without introducing any unexpected behaviors. We also formalized the attacker model in this chapter, where we formalized the attacker's capabilities in terms of observations and operations. The verification tools that we later develop in Chapter 7, Chapter 8, and Chapter 9 provide formal assurances against these attacker models.

# Chapter 5

# Formal Verification of Secure Remote Execution on Enclave Platforms

In Chapter 3, we developed a formal semantics for reasoning about enclave execution, while assuming that the platform guarantees security properties in the presence of a privileged adversary. When the user outsources an enclave to a remote platform, she seeks a guarantee that the enclave be executed according to the expected behavior. This chapter discharges the security assumption we make on the platform, by verifying that the platform satisfies *secure remote execution (SRE)*: any execution of that enclave on the platform must be one of enclave's behaviors (formalized in Equation 3.1). We formalize the SRE property and prove that several popular enclave platforms, specifically Intel SGX and Sanctum, satisfy SRE. The contributions of this chapter are central to our thesis goal of certifying confidentiality properties of enclave programs with (nearly) zero trusted computing base.

First, we show how SRE can be decomposed into lower-level properties — specifically, integrity, confidentiality, and secure measurement. Then, we prove these three properties on the TAP model, while varying the adversary's capabilities (recall adversaries $M$, $MC$, $MP$, and $MCP$ from Section 4.2). Since we proved in Chapter 4 that SGX and Sanctum models refine the TAP, we automatically get that they also satisfy the three properties of SRE (albeit under different adversaries), and therefore, SRE itself.

This chapter is structured as follows. We define SRE in Section 5.1, and discuss its decomposition into integrity, confidentiality, and measurement properties in Section 5.2 and Section 5.3. Section 5.4 discusses the usefulness of SRE i.e. what properties of secure computation it does and does not cover, and how to build upon SRE to develop applications with end-to-end security properties. We prove that TAP satisfies SRE in Section 5.5, and present various empirical results of the verification effort in Section 5.6.

## 5.1 Secure Remote Execution of Enclaves

Until now, we defined the semantics of enclave execution assuming a safe enclave platform. In this section, we formally specify this safety property, termed *secure remote execution* (SRE), and discuss a method to prove that a remote platform provides the SRE guarantee.

Imagine a user who wishes to outsource the execution of an enclave program $e$ onto a remote platform. The user desires that the platform respect the enclave's semantics by executing trace(s) from $[\![e]\!]$ (defined in Equation 3.1). Since the privileged software layers on the platform are untrusted, the user's trust is based on guarantees provided by the hardware platform. We propose the following notion of secure remote execution (SRE) of enclaves:

**Definition 2** *Secure Remote Execution of Enclaves. A remote platform performs secure execution of an enclave program $e$ if any execution trace of $e$ on the platform is contained within $[\![e]\!]$. Furthermore, the platform must guarantee that a privileged software attacker only observes a projection of the execution trace, as defined by the observation function obs.*

It is important to note that SRE does not force the platform to execute $e$ — the attacker may deny service, and this is easily detectable by the user because the attacker cannot forge attested statements as if they originated from the user's enclave. Nor are we forcing the platform to execute $e$ a fixed number of times. The attacker has the capability to execute $e$ as many times as it wishes (with the hope of performing dictionary attacks, for example to learn secret data), and a user can easily defend against these attacks by refusing to provision secrets to other copies of the enclave. With that said, SRE requires the platform to execute a trace from $[\![e]\!]$, and recall that $[\![e]\!]$ only contains enclave executions that start in the initial state of the enclave (see Equation 3.1). Furthermore, recall that our definition of $[\![e]\!]$ assumes secure execution of $e$ in that the attacker only affects $e$'s execution by affecting the inputs, which are assumed to be untrusted anyway — we later state an integrity component of SRE that validates this assumption of the enclave platform.

## 5.2 Proof Decomposition of SRE

A rational user will outsource the enclave only to a platform that provides a formal guarantee of SRE. To that end, we describe a method for formally verifying that an enclave platform provides SRE to any enclave program; we later develop machine-checked proofs that SGX and Sanctum satisfies SRE. The key idea is to decompose the SRE property into the following three lower-level properties:

- **Secure Measurement**: The platform must *measure* the enclave program to enable the user to detect any changes to $e$'s launch state $init_e$ and configuration $config_e$ i.e. the user must be able to verify that the platform is running an unmodified $e$.
- **Integrity**: The enclave program's execution cannot be affected by a privileged software attacker beyond providing inputs, i.e. the sequence of inputs uniquely determines the enclave's execution trace, and that trace must be contained within $[\![e]\!]$.

- **Confidentiality**: A privileged software attacker does not observe any aspect of enclave execution beyond what is already revealed by *obs*. In other words, the attacker cannot distinguish between two executions that have equivalent observations *obs*.

The measurement and integrity properties guarantee that the remote platform executes a trace from $[\![e]\!]$, while the confidentiality property ensures that the attacker does not learn more information than what the enclave wishes to reveal. Together, these three properties imply SRE, and we formalize each one of them in the remainder of this section.

### 5.2.1 Secure Measurement

During launch of an enclave $e$, the platform computes a hash of $e$'s initial contents ($init_e$) along with relevant configuration bits ($config_e$). The hash-based measurement acts as a unique identity for the enclave, which follows directly from the collision resistance assumption of the cryptographic hash function, and the platform includes this measurement in each attested statement. Any deviation from the desired enclave program will be detected when the enclave sends an attested statement to the user — we assume that attested statements are produced using a quoting scheme that is unforgeable under chosen message attacks (UF-CMA); we do not model the cryptography of this scheme, and refer the reader to [93] for a formal treatment of this subject. The secure measurement property states that any two enclaves with the same measurement must also have the same semantics: they must produce equivalent execution traces for equivalent input sequences.

Let $\mu(e)$ be the measurement of enclave $e$, performed when launching the enclave. The measurement function must be such that two enclaves with the same measurement value have identical states.

$$\begin{aligned} \forall \sigma_1, \sigma_2.\ & init_{e_1}(E_{e1}(\sigma_1)) \wedge init_{e_2}(E_{e2}(\sigma_2)) \Rightarrow \\ & \mu(e_1) = \mu(e_2) \iff E_{e_1}(\sigma_1) = E_{e_2}(\sigma_2) \end{aligned} \tag{5.1}$$

Next we need to ensure that if two enclaves $e_1$ and $e_2$ have the same state, then they produce equivalent execution traces for equivalent input sequences. This is the determinism property we assumed while defining $[\![e]\!]$ in Section 3.6 of the enclave platform, so we must prove that it holds on the enclave platform.

$$\begin{aligned} & \forall \pi_1, \pi_2. \\ & \big(E_{e_1}(\pi_1[0]) = E_{e_2}(\pi_2[0]) \qquad \wedge \\ & \quad \forall i.\ (curr(\pi_1[i]) = e_1) \iff (curr(\pi_2[i]) = e_2) \qquad \wedge \\ & \quad \forall i.\ (curr(\pi_1[i]) = e_1) \implies I_{e_1}(\pi_1[i]) = I_{e_2}(\pi_2[i])\big) \implies \\ & \big(\forall i.\ E_{e_1}(\pi_1[i]) = E_{e_2}(\pi_2[i]) \wedge O_{e_1}(\pi_1[i]) = O_{e_2}(\pi_2[i])\big) \end{aligned} \tag{5.2}$$

Consider two enclaves $e_1$ and $e_2$ that produce execution traces $\pi_1$ and $\pi_2$, respectively. Equation 5.2 states that if: (i) $\pi_1$ and $\pi_2$ start with the same initial state, (ii) $\pi_1$ and $\pi_2$

enter and exit the enclaves in lockstep (i.e. the two enclaves execute the same number of steps), and (iii) if the input sequences in $\pi_1$ and $\pi_2$ are equivalent (i.e. $I_{e_1}(\pi_1[i]) = I_{e_2}(\pi_2[i])$), then the two enclaves execute identically in both traces: they have the same sequence of state and output values (i.e. $E_{e_1}(\pi_1[i]) = E_{e_2}(\pi_2[i]) \wedge O_{e_1}(\pi_1[i]) = O_{e_2}(\pi_2[i])$).

### 5.2.2 Integrity

The integrity guarantee ensures that the execution of the enclave in the presence of attacker operations is identical to the execution of the program without the attacker's operations. In other words, the attacker only impacts an enclave's execution by controlling the sequence of inputs — all other operations, such as controlling I/O peripherals and executing supervisor-mode instructions, have no effect on the enclave's execution. Any two traces (of the same enclave program) that start with equivalent enclave states and have the same input sequence will produce the same sequence of enclave states and outputs, even though the attacker's operations may differ in the two traces.

$$
\begin{aligned}
&\forall \pi_1, \pi_2. \\
&\big(E_e(\pi_1[0]) = E_e(\pi_2[0]) \qquad \wedge \\
&\ \forall i.\ (curr(\pi_1[i]) = e) \iff (curr(\pi_2[i]) = e) \qquad \wedge \\
&\ \forall i.\ (curr(\pi_1[i]) = e) \implies I_e(\pi_1[i]) = I_e(\pi_2[i])\big) \implies \\
&\big(\forall i.\ E_e(\pi_1[i]) = E_e(\pi_2[i]) \wedge O_e(\pi_1[i]) = O_e(\pi_2[i])\big)
\end{aligned}
\tag{5.3}
$$

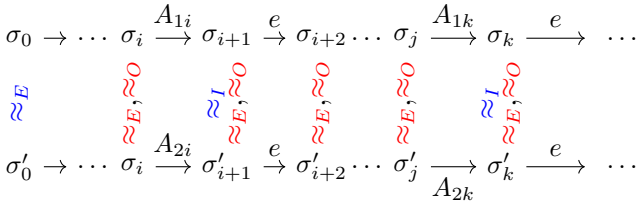



*Figure 5.1*: Integrity property.

Figure 5.1 shows the two traces from the integrity property. The adversary's steps are labelled $A$ and enclave's steps are labelled $e$. Assumptions are annotated in blue, and proof obligations are shown in red. The enclave's inputs are assumed to be the same in both traces; this is shown by the $\approx_I$ symbol. The initial state of the two enclaves is assumed to be the same. The attacker performs different actions in each trace, but the integrity proof must show that the enclave's state and outputs do not differ: $\forall i.\ E_e(\pi_1[i]) = E_e(\pi_2[i]) \wedge O_e(\pi_1[i]) = O_e(\pi_2[i])$. These proof obligations are denoted by the red $\approx_E$ and $\approx_O$ symbols.

### 5.2.3 Confidentiality

The enclave platform must ensure that the attacker does not observe the enclave's execution beyond what is allowed by the observation function *obs*. For any enclave platform, *obs* must, at the very least, include the initial memory *init*, configuration *config*, outputs to non-enclave memory, exit events from enclave mode to untrusted code — depending on the platform, *obs* may also leak certain side channels e.g. page-level access patterns on SGX. The privileged software attacker may use any combination of machine instructions to perform an attack, and the attacker can trivially distinguish between two different enclaves based on the differences in *obs* e.g. if the two executions produce different outputs. However, confidentiality states that no other information is leaked: two enclave executions that produce the same observation (but with potentially differently enclave states) must be indistinguishable to the attacker. For confidentiality to hold, the attacker must observe the same results from executing the adversarial instructions in both traces — since the attacker has the same sequence of states in both traces, we say that the attacker has not observed any information about the enclave beyond what is revealed in *obs*. If the platform contains a vulnerability that allows the attacker to observe different values in the two traces (with the same observation), then attacker's states would diverge in the two traces, and we say that such a platform does not provide confidentiality.

$$
\begin{aligned}
&\forall \pi_1, \pi_2. \\
&\big(A_{e1}(\pi_1[0]) = A_{e2}(\pi_2[0]) \qquad \wedge \\
&\ \forall i.\ curr(\pi_1[i]) = curr(\pi_2[i]) \wedge I^P(\pi_1[i]) = I^P(\pi_2[i]) \qquad \wedge \\
&\ \forall i.\ curr(\pi_1[i]) = e \implies obs_{e1}(\pi_1[i+1]) = obs_{e2}(\pi_2[i+1])\big) \implies \\
&\ \big(\forall i.\ A_{e1}(\pi_1[i]) = A_{e2}(\pi_2[i])\big)
\end{aligned}
\tag{5.4}
$$

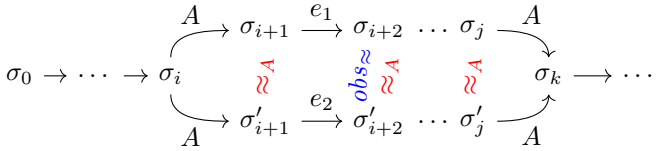



*Figure 5.2*: Confidentiality property.

Figure 5.2 shows the two traces in the confidentiality property. As in Figure 5.1, the attacker's steps are labelled $A$ and the enclave's steps are labelled $e$. The two traces start off in equivalent states but diverge (at state $\sigma_i$) because the two *enclaves* may perform different computation. The enclave's adversary-visible *observations* are assumed to be the same in

both traces when the enclave is executing; this is shown by the blue $obs_{\approx}$. The platform's non-determinism ($I^P(\sigma)$) is also assumed to be the same in both traces, otherwise the two traces would diverge because of external inputs rather than leakage of enclave's state. The two traces eventually merge (to the same platform state) when the enclave is destroyed. The theorem states that adversary state is identical: $\forall i.\ A_{e1}(\pi_1[i]) = A_{e2}(\pi_2[i])$; this is illustrated in the red $\approx_A$. We conclude that the attacker has not learned any additional information beyond *obs*.

## 5.3 Soundness of SRE Decomposition

**Theorem 1** *An enclave platform that satisfies secure measurement, integrity, and confidentiality property for any enclave program also satisfies secure remote execution.*

**Proof**: Suppose that the user sends an arbitrary enclave $e$ to a remote server for execution, and the platform launches enclave $e_r$ some time later — because $e$ is sent over an untrusted channel, $e$ may or may not equal $e_r$. If the user finds $\mu(e_r) \neq \mu(e)$, then the platform has no obligations to execute a trace from $[\![e]\!]$. Otherwise, if $\mu(e_r) = \mu(e)$, we derive $[\![e_r]\!] = [\![e]\!]$, thanks to the measurement (Equation 5.1, Equation 5.2) and integrity properties (Equation 5.3), as per the following explanation. Equation 5.1 implies that $E_{e_r}(\pi_r[0]) = E_e(\pi[0])$ for any $\pi_r \in [\![e_r]\!], \pi \in [\![e]\!]$, i.e., equivalent hash-based measurement implies equivalent initial enclave state (ignoring hash collisions). Next, Equation 5.2 implies that for any such pair of traces $\pi_r$ and $\pi$ that have equivalent initial states, if the inputs along the traces are equal, then the state transitions and outputs along the traces are also equal, i.e. we derive $\forall i.\ (I_{e_r}(\pi_r[i]), E_{e_r}(\pi_r[i]), O_{e_r}(\pi_r[i])) = (I_e(\pi[i]), E_e(\pi[i]), O_e(\pi[i]))$ from $\forall i.\ I_{e_r}(\pi_r[i]) = I_e(\pi[i])$. Consequently, for each trace $\pi_r \in [\![e_r]\!]$, we have a trace $\pi \in [\![e]\!]$ whose inputs, outputs, and states are equivalent, and vice versa; in other words, $[\![e_r]\!] = [\![e]\!]$. Therefore, in the case $\mu(e_r) = \mu(e)$, we prove that the two programs $e_r$ and $e$ have identical runtime behaviors, which is a prerequisite for SRE. Finally, confidentiality (Equation 5.4) implies that the attacker's observation is restricted to *obs*.

## 5.4 Application of Secure Remote Execution

SRE is an important stepping stone for building practical applications with security guarantees. Consider the problem of executing a batch job securely in the cloud. The user sends an enclave program, which implements a function on sensitive data, to an enclave platform in the cloud. The protocol includes the following steps:

1. The user sends an enclave program $e$ to the cloud provider, which launches the program on an enclave platform.

2. The user and enclave establish an authenticated TLS channel via an ephemeral Diffie-Hellman (D-H) exchange.

(a) User sends her public parameter $g^x$ to the enclave, where $x$ is a randomly generated fresh value.

(b) Enclave sends its public parameter $\texttt{attest}(g^y)$ to the user, in the form of an attested statement, thus guaranteeing that a genuine enclave platform launched the expected enclave $e$.

(c) User and enclave compute a shared master secret $g^{xy}$, and derive symmetric session keys $s_{ue}$ and $s_{eu}$, a key for each direction.

3. The user now sends encrypted input to enclave using this shared secret: $\{in\}_{s_{ue}}$.

4. The enclave decrypts its input, performs the computation and returns the encrypted result to the user: $\{out\}_{s_{eu}}$.

Consider the following security property: the attacker neither learns secret input $\{in\}$ nor the secret output $\{out\}$. To that end, the user 1) develops an enclave program that only accepts encrypted inputs and sends encrypted outputs, and 2) specifies an observation function (*obs*) where a privileged software adversary is only allowed to view the enclave's outputs to non-enclave memory — this is acceptable because $e$ encrypts its outputs.

The measurement guarantees that the user will only establish a channel with the expected enclave on a genuine enclave platform. Integrity ensures that the platform will execute a trace from $[\![e]\!]$, thus respecting $e$'s semantics. The platform may choose to not launch $e$ or prematurely terminate $e$, but such executions will not generate $\{out\}_{s_{eu}}$ and hence can be trivially detected by the user. Integrity also ensures that the platform does not rollback the contents of enclave's memory while it is alive (i.e. not destroyed) as such attacks will cause the enclave's execution to proceed differently from $[\![e]\!]$, and SRE guarantees $[\![e]\!]$. SRE does not require the platform to defend against rollback attacks on persistent storage. This protection is not needed for the batch service because the enclave does not update $\{in\}_{s_{ue}}$, and any tampering to $\{in\}_{s_{ue}}$ will fail the cryptographic integrity checks. Finally, confidentiality ensures that the enclave platform only reveals the *obs* function of enclave's execution to the software attacker, which only includes the encrypted outputs. We now have our end-to-end security property.

Should the enclave require state beyond enclave's memory to perform the job, it would require integrity, freshness, and confidentiality for non-enclave state, which is not covered by SRE. The enclave can implement cryptographic protections (e.g. Merkle tree) and techniques for state continuity [92] to address this concern.

## 5.5 Proof of Secure Remote Execution for TAP

We proved three machine-checked theorems that correspond to the requirements for secure remote execution as described in Section 5.2.

**TAP Integrity**: We proved that the integrity result (Equation 5.3) holds for the TAP for all three adversaries: $M$, $MC$ and $MCP$. It shows that these adversaries have no effect on enclave execution beyond providing inputs via non-enclave memory.
**TAP Measurement**: We showed that Equation 5.1 and Equation 5.2 are satisfied by the TAP. The proof for Equation 5.2 need not include adversarial operations because integrity ensures that an adversary cannot affect enclave's execution beyond providing inputs.
**TAP Confidentiality**: We showed three confidentiality results, each corresponding to the three TAP adversaries: $M$, $MC$, and $MCP$.

Confidentiality holds unconditionally for adversary $M$.

For adversary $MC$, let $pa2set : \text{PA} \rightarrow \text{Set}$ be the function that maps physical addresses to cache sets. This function is uninterpreted (abstract) in the TAP and will be defined by implementation. We showed that confidentiality holds for adversary $MC$ if Equation 5.5 is satisfied: a physical address belonging to an enclave never shares a cache set with a physical address outside the enclave.

$$\forall p_1, p_2, \sigma, e. \qquad (5.5)$$
$$\sigma(\texttt{owner}[p_1] = e \wedge \texttt{owner}[p_2] \neq e) \implies (pa2set(p_1) \neq pa2set(p_2))$$

Finally, we showed that confidentiality holds for adversary $MCP$ if Equation 5.5 is satisfied by the TAP implementation and the TAP configuration Boolean `priv_mappings` is `true`.

## 5.6 Verification Results

This section discusses our models and machine-checked proofs. Our models of the TAP, Intel SGX and MIT Sanctum are constructed using the BoogiePL [14] intermediate verification language. BoogiePL programs can be annotated with assertions, pre-conditions and post-conditions for procedures and loop invariants. The validity of these annotations are checked using the Boogie verification condition generator [15], which in turn uses automated theorem provers like the Z3 SMT solver [41]. Procedure pre- and post-conditions, in particular, enable modular verification using Boogie. For example, we specify the behavior of each TAP operation using pre- and post-conditions and verify that the implementation of these procedures satisfies these post-conditions. Once they are verified, TAP's proofs of integrity, confidentiality and secure measurement can soundly ignore the implementation and only reason about the pre- and post-conditions of these operations.

### 5.6.1 BoogiePL Model Construction

The modeling and verification effort spanned eight person months, of which three person months were spent on modeling alone, while the remaining five months were spent on the verification effort, which mostly incorporated diagnosing proof failures and brainstorming additional invariants to guide the theorem prover to produce the proofs. Overall, we found

that the theorem prover (Z3 SMT solver) required substantial annotations (in the form of invariants, preconditions, postconditions, and intermediate lemmas) to verify the proof, and further research is needed in automatically synthesizing these annotations to reduce the manual burden.

| **Description** | **Size** | | | | **Verif. Time (s)** |
|---|---|---|---|---|---|
| | #pr | #fn | #an | #ln | |
| TAP | 22 | 49 | 204 | 1752 | 5 |
| Integrity | 12 | 13 | 145 | 985 | 26 |
| Measurement | 6 | 3 | 100 | 800 | 6 |
| Confidentiality | 8 | - | 200 | 1388 | 194 |
| MMU Model | 9 | 13 | 68 | 739 | 7 |
| MMU Refinement | 3 | 2 | 38 | 216 | 8 |
| Sanctum | 23 | 321 | 44 | 780 | 1 |
| Sanctum Refinement | 12 | 3 | 94 | 548 | 11 |
| SGX | 36 | 113 | 4 | 1526 | - |
| SGX Refinement | 10 | 1 | 38 | 351 | 2 |
| **Total** | **141** | **518** | **935** | **9085** | **260** |

*Table 5.1*: BoogiePL Models and Verification Results

Table 5.1 shows the approximate size of each of our models. #pr, #fn, #an and #ln refers to the number of procedures, functions, annotations and lines of code respectively. Annotations refer to the number of loop invariants, assertions, assumptions, pre- and post-conditions that we manually specify. While Boogie can discharge some assertions automatically, we found that we had to manually specify 935 annotations before it accepted our proofs.

The rows TAP, MMU Model, Sanctum and SGX correspond to models of the functionality of the TAP, the Sanctum MMU, Sanctum, and SGX respectively. The other rows correspond to our proofs of SRE and refinement. In total, the models are about 4800 lines of code while the proofs form the remaining $\approx$ 4300 lines of code. A significant part of the effort in developing the proofs was finding the correct invariants to help Boogie prove the properties.

BoogiePL can only verify *safety* properties. But many of our theorems involve *hyper-properties* [36]. We used the self-composition construction [18, 118] to convert these into safety properties. BoogiePL is also incapable of verifying properties involving alternating nested quantifiers, for example, $\forall x.\ \exists y.\ \phi(x, y)$. We *skolemized* [114] such properties to take the form: $\forall x.\ \forall y\ .(y = f(x)) \implies \phi(x, y)$; $f(x)$ is called a *Skolem function* and must be manually specified.

### 5.6.2 Verification Results

Table 5.1 lists the total time taken by Boogie/Z3 to check validity of all the manually specified annotations — by verifying the annotations, we omit them from the trusted computing base, which only includes the Boogie/Z3 theorem prover. The verification times for the TAP, MMU Model and Sanctum rows is for proving that procedures in these models satisfy their post conditions, which specify behavior and system invariants. The verification times for the remaining rows is the time taken to prove the SRE properties and refinement. The total computation time in checking validity of the proofs once the correct annotations are specified is only a few minutes.

## 5.7 Summary

This chapter formalized the guarantee expected from an enclave platform during the execution of any enclave program. We call this property of the platform secure remote execution (SRE), and perform proofs demonstrating that TAP satisfies the three properties required for SRE: secure measurement, integrity and confidentiality. Since we already proved (Chapter 4) that Intel SGX and Sanctum are refinements of TAP under certain adversarial conditions, we derive that Intel SGX and Sanctum also satisfy the properties required for SRE. Overall, this chapter takes an important step towards a unified, extensible framework for reasoning about enclave programs and platforms.

# Part II

# Secure Enclaves: Design and Verification

# Foreword on Part II

The primitives offered by trusted enclave platforms are necessary for any form of trustworthy computing on sensitive data, but not sufficient by themselves. The enclave programs must also be secure, which includes, at the very least, a provable guarantee that sensitive data is never leaked to the attacker in any execution, under any potential attack scenario — we refer to this property of the enclaves as confidentiality, which is complementary to the confidentiality guarantee offered by the enclave platform (recall the requirements of secure remote execution from Chapter 5). However, the burden of programming enclaves correctly and ensuring confidentiality remains with the programmer. Guaranteeing that enclave programs satisfy confidentiality has several challenges, which we address in this part of the thesis.

Recall the formal definition of a privileged software adversary in Chapter 4, where we define several variants of the adversary (namely $M$, $MC$, $MP$, and $MCP$), all of which observe and modify non-enclave memory at all times during the enclave's execution. Since, an enclave will undoubtedly interact with the external world (via non-enclave memory) to receive inputs and produce outputs, it gives the adversary a mechanism to interact with the enclave and exploit any of its vulnerabilities. For instance, since the enclave fetches its inputs by reading from non-enclave memory, the attacker can modify non-enclave memory at any time and perform time-of-check-to-time-of-use style attacks on an insecure enclave. Furthermore, since the attacker can access any outputs written to non-enclave memory, the enclaves should encrypt secrets before writing them to non-enclave memory — however, the programmer may make memory safety errors such as buffer overrun, dangling pointer dereferences, etc., which may trigger subtle vulnerabilities in the enclave and write out cleartext values. Writing safe enclaves that avoid such attacks is non-trivial, and prior experience suggests that non-expert developers often make critical errors such as memory corruption bugs [117], unsafe use of cryptographic operations, insufficient access control, information flow leaks, etc. Chapter 7 and Chapter 8 present methodologies and automatic tools to both detect and prevent such vulnerabilities.

To add to the developer's woes, some mainstream enclave platforms do not provide idealized execution i.e. while they may offer isolated execution, they do not necessarily offer strict confidentiality — recall adversaries $MP$ and $MCP$ defined in Chapter 4. For instance, Intel SGX leaks information about the enclave's execution via various software and hardware side channels (e.g. cache timing attacks, page-level memory access patterns), and protecting these leaks requires even greater understanding of hardware, careful programming practices, and tool support for automatically enforcing and verifying defenses. Closing the side channel of page-level access patterns is the focus of Chapter 9, which presents a toolchain consisting of a compiler and verifier to automatically produce machine code that hides such implicit leaks.

In summary, this part of the thesis presents a novel methodology for safely programming enclaves and formally certifying the absence of vulnerabilities that leak secrets, both via explicit outputs and certain side channels.

# Chapter 6

# Formalizing Confidentiality

Prior to describing tools that certify confidentiality properties, we use this chapter to formalize confidentiality of enclave programs, with respect to both outputs and side channels. Confidentiality can be expressed as an information-flow policy that tracks the flow of secrets through the program, and checks whether secrets may explicitly or implicitly flow to some state that the adversary can observe [22, 43, 23, 87].

This chapter has the following organization. Section 6.1 describes a transformation on our model of the enclave program that models the effect of a privileged software adversary on the enclave's execution. Section 6.2 formalizes confidentiality with respect to an enclave's outputs (i.e. writes to non-enclave memory). Finally, Section 6.3 extends the confidentiality definition from Section 6.2 to also cover side channels, specifically the page-level memory access pattern which we mitigate in Chapter 9.

## 6.1 Modeling Adversary's Effect on Enclave Execution

We defined the enclave's model of execution in Section 3.6, where we formalized an enclave program as a state transition system. Recall that an enclave $e$ computes in steps, with each step executing a machine instruction (including trusted primitives provided by the enclave platform), where the attacker provides an input ($I_e^U(\sigma)$) at each step via non-enclave memory. Although we assumed in this definition that this is the only mechanism for an attacker to affect an enclave's execution, we discharge this assumption in Chapter 5, where we prove an integrity theorem that a TAP attacker that performs an unbounded number of allowed adversarial operations can be simulated by some update to (all addresses in) non-enclave memory. To model this attacker that provides a fresh input $I_e^U(\sigma)$ at each step, we introduce a havoc $\texttt{mem}_{\neg\texttt{evrange}}$ operation that clobbers all locations not in `evrange` (i.e. outside enclave memeory) with an arbitrary value — note that all of our adversaries ($M$, $MC$, $MP$, and $MCP$) have full access to non-enclave memory, and consequently, the ability to perform havoc $\texttt{mem}_{\neg\texttt{evrange}}$. Since the attacker can execute at any time, we instrument this operation before any enclave instruction. By interleaving havoc $\texttt{mem}_{\neg\texttt{evrange}}$ operations with enclave's instructions, we capture all potential effects of the attacker on the enclave's executions. In

other words, we soundly model all potential executions of the enclave in the presence of a privileged software attacker, and the proof of secure remote execution in Chapter 5 backs this claim.

We now precisely describe the transformation on the enclave program model. For the reader's convenience, we first recall the relevant formalism from Section 3.6. An execution trace of the platform is an unbounded-length sequence of states denoted $\langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$, such that $\forall i.\ \sigma_i \rightsquigarrow \sigma_{i+1}$. An enclave $e$ is only executing in some of the steps of the platform's trace, and we write that sub-sequence as $\pi = \langle \sigma'_0, \sigma'_1, \ldots, \sigma'_m \rangle$, where $init_e(E_e(\sigma'_0)) \wedge \forall i.\ curr(\sigma'_i) = e$. To model the effect of a privileged adversary (e.g. $M$), for any execution trace $\pi$ of $e$, we interleave $\mathsf{havoc}\ \mathtt{mem}_{\neg \mathtt{evrange}}$ operations, thus giving us the trace $\langle \widehat{\sigma_0}, \sigma'_0, \widehat{\sigma_1}, \sigma'_1, \ldots, \widehat{\sigma_m}, \sigma'_m \rangle$ where $init_e(E_e(\sigma'_0)) \ \wedge\ \forall i.\ curr(\sigma'_i) = e \ \wedge\ \forall i.\ (\widehat{\sigma_i}, \sigma'_i) \in \mathsf{havoc}\ \mathtt{mem}_{\neg \mathtt{evrange}}$ — treat $\mathsf{havoc}\ \mathtt{mem}_{\neg \mathtt{evrange}}$ as a relation that relates any pair of states that have equivalent values of enclave $e$'s memory (within $\mathtt{evrange}$), but arbitrary values elsewhere ($\neg\mathtt{evrange}$).

Furthermore, in order to include $\mathsf{havoc}\ \mathtt{mem}_{\neg \mathtt{evrange}}$ operations for any execution trace of an enclave, we define the following instrumentation. Given an enclave program $p$, which is a sequence of machine instructions $i; \ldots; i$, we instrument $\mathsf{havoc}\ \mathtt{mem}_{\neg \mathtt{evrange}}$ prior to each instruction $i$ in $p$. The resulting program is $p_M = \mathsf{havoc}\ \mathtt{mem}_{\neg \mathtt{evrange}}; i; \ldots; \mathsf{havoc}\ \mathtt{mem}_{\neg \mathtt{evrange}}; i$ — we choose the notation $p_M$ to mean program $p$ instrumented with the operations of an attacker $M$. This instrumentation guarantees that any execution trace of $p_M$ interleaves $\mathsf{havoc}\ \mathtt{mem}_{\neg \mathtt{evrange}}$ operations with the enclave's instructions.

## 6.2 Confidentiality

We wish to guarantee that an enclave program does not leak its secret state to attacker-visible state, by proving that an attacker's observation does not include the enclave's secrets. More precisely, an attacker's observation, during an enclave program's execution, must be independent of the enclave's secrets. We refer to this property of an enclave program as confidentiality. This notion of confidentiality can be formulated as a non-interference property [81], specifically in the form of observational determinism [36]: the attacker's observation is a deterministic function of the attacker's operations, and nothing else. The remainder of this section is devoted to formalizing this definition of confidentiality for enclave programs.

From here on, we assume that the enclave program is already instrumented with the $\mathsf{havoc}\ \mathtt{mem}_{\neg \mathtt{evrange}}$ operations to model the effect of a privileged software attacker, as we describe in Section 6.1. An execution trace $\pi$ starts in the initial state of the machine following a power cycle; at some point in the trace, the adversary launches the enclave program. From then on, $\pi$ consists of alternating sequences of adversarial and enclave instructions. We use $seq_A(\pi, i)$ and $seq_E(\pi, i)$ to denote the i-th subsequence of adversarial and enclave instructions, respectively; Figure 6.1 illustrates these functions. For the reader's convenience, we now recall some notation that we developed in Section 3.6. Let the projection function $A$ denote the component of enclave-observable machine state that the adversary is allowed to control; we define $A(\sigma) = \sigma(\mathtt{mem}_{\neg \mathtt{evrange}})$, where $\mathtt{mem}_{\neg \mathtt{evrange}}$ denotes non-enclave

memory. Note that the adversary may invoke privileged instructions that modify state beyond $\texttt{mem}_{\neg\texttt{evrange}}$ (e.g. control register CR4). However, we omit these state variables from $A$ because they are not included in the enclave's state or the operational semantics, and the trusted CPU must guarantee the operational semantics during enclave execution, regardless of how the adversary manipulates this additional machine state — we prove the integrity theorem in Chapter 5 that establishes exactly this guarantee.

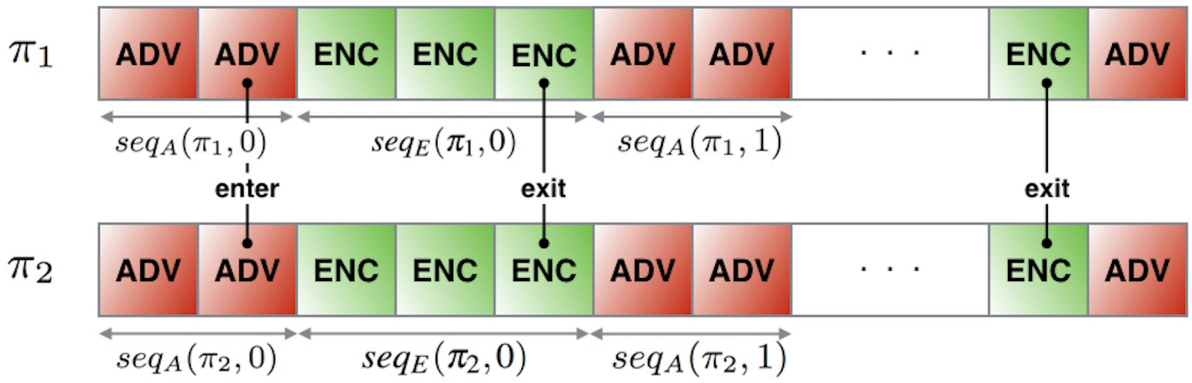



*Figure 6.1*: Illustration of confidentiality definition. The untrusted platform transfers control to the enclave by invoking enter, and enclave transfers control back by invoking exit.

**Definition 3 Confidentiality** *For any pair of execution traces of the enclave $e$, if the adversary's operations along the two traces are equivalent, then the adversary's observations along the two traces must also be equivalent.*

$$\forall \pi_1, \pi_2 \in [\![e]\!].\pi_1 \equiv_A \pi_2 \Rightarrow \pi_1 \equiv_O \pi_2 \tag{6.1}$$

*where*

$$\pi_1 \equiv_A \pi_2 \Leftrightarrow \forall i.\ \texttt{instr}(seq_A(\pi_1, i)) \equiv \texttt{instr}(seq_A(\pi_2, i)) \ \wedge \ A(seq_E(\pi_1, i)[0]) \equiv A(seq_E(\pi_2, i)[0])$$

*and*

$$\pi_1 \equiv_O \pi_2 \Leftrightarrow \forall i.\ obs_e^M(\pi_1[i]) \equiv obs_e^M(\pi_2[i])$$

Confidentiality, a hyper-property defined over pairs of executions, is violated when the enclave produces observationally different traces for equivalent adversarial operations. Equivalence is defined using relations $\equiv_A$ and $\equiv_O$. The equivalence relation $\equiv_A$ over pairs of adversarial subsequences ($seq_A(\pi_1, i)$ and $seq_A(\pi_2, i)$) only includes traces that 1) have equal lengths, 2) have the same instructions, and 3) produce equivalent sequence of states, where equivalence over states is defined modulo the projection function $A$. Equality of states modulo $A$ is naturally defined to be bitwise equivalence for all locations in $\texttt{mem}_{\neg\texttt{evrange}}$, with the caveat that encrypted values can differ. Specifically, using the approach of cryptographically-masked flows [12], we treat all valid ciphertexts to be equivalent — this ensures that fetching encrypted secret inputs will produce potentially different secret inputs in both $\pi_1$ and $\pi_2$,

and allow us to reason about whether the attacker can distinguish between observations of traces $\pi_1$ and $\pi_2$ with different values of secrets. Without this restriction, the antecedent of the logical property in definition 3 would force all encrypted inputs to have equivalent bitwise values, thereby forcing secrets to have the same values in $\pi_1$ and $\pi_2$, which may let the enclave trivially satisfy confidentiality (because no difference is observed in $\pi_1$ and $\pi_2$). For checking that the enclave's observed behaviors are equivalent, we use an equivalence relation $\equiv_O$, defined simply using the $obs^M$ from Section 4.2. As we stated before, this definition of confidentiality is a form of non-interference property, adapted to the enclave model of execution. We point out that this definition does not prevent leaks via the timing channel because we model an enclave's execution as a state transition system, abstracting away any timing information — each state transition corresponds to a processor instruction. This definition of confidentiality also allows leaks information via the termination channel, thus making it termination-insensitive non-interference [64].

## 6.3 Page Access Obliviousness

By simply extending the observation function *obs* to cover side channel observations, in addition to the contents of non-enclave memory, we extend the confidentiality definition 3 to account for side channel leaks. Specifically, we focus on the side channel of page-level memory access pattern, which is exposed on some enclave platforms, specifically SGX — defending against leaks via this channel is discussed in Chapter 9. In this section, we formalize a variant of confidentiality that states that the enclave must not produce different observations of page-level access patterns due to different secret state. We term this property *page access obliviousness*: the enclave's page-level memory accesses is independent of its secrets.

**Definition 4 Page Access Obliviousness** *An enclave is page access oblivious if it satisfies confidentiality, with observation function $O$ set to the following $PA$ function*[1]*.*

$$\pi_1 \equiv_O \pi_2 \Leftrightarrow$$
$$PA(seq_E(\pi_1, i)) \equiv PA(seq_E(\pi_2, i))$$

$$PA(\sigma_0, \ldots, \sigma_n) \doteq [PA_c(\sigma_0) \cdot PA_d(\sigma_0) \cdot \ \ldots \ \cdot PA_c(\sigma_n) \cdot PA_d(\sigma_n)]$$
$$PA_c(\sigma) \doteq \langle execute, \sigma(\mathsf{rip}/2^{\mathsf{p}})\rangle$$

$$PA_d(\sigma) \doteq \begin{cases} \langle read, \sigma(\mathsf{reg_a}/2^{\mathsf{p}})\rangle & \text{if } \mathtt{instr}(\sigma) = \mathsf{mov\ reg_d\ [reg_a]} \\ \langle write, \sigma(\mathsf{reg_a}/2^{\mathsf{p}})\rangle & \text{if } \mathtt{instr}(\sigma) = \mathsf{mov\ [reg_a]\ reg_d} \\ \langle read, \sigma(\mathsf{rsp}/2^{\mathsf{p}})\rangle & \text{if } \mathtt{instr}(\sigma) \in \{\mathsf{pop\ reg},\ \mathsf{ret}\} \\ \langle write, \sigma(\mathsf{rsp}/2^{\mathsf{p}})\rangle & \text{if } \mathtt{instr}(\sigma) \in \{\mathsf{push\ reg},\ \mathsf{call}\} \\ \epsilon & \text{otherwise} \end{cases}$$

[1] *Note that $\cdot$ operator denotes list concatenation.*

The $PA$ function permits the adversary to observe all memory accesses at the page-level granularity (enforced by the division by page size $2^p$). The value of p is architecture-specific; a page has size 4096 bytes in Intel SGX CPUs, which makes p = 12. As dictated by $PA$, for each instruction executed in enclave-mode, the attacker records 1) access to a code page at address rip to fetch the instruction, and 2) access to a data page, if the instruction triggers a data access — for instance, push performs a write access to a data page at address rsp. This definition of $PA$ represents the x86-64 ISA semantics, even in the presence of optimizations such as prefetching and caching, because the CPU evaluates the page permission check for each memory access. The adversary also observes the type of memory access: read, write, or execute.

## 6.4 Summary

This chapter defined confidentiality properties of enclave programs, with respect to both explicit outputs (via writes to non-enclave memory), and side channel observations (specifically page-level access pattern). The verification techniques that we develop next in Chapter 7 and Chapter 8 guarantee the confidentiality property of explicit outputs that we formalize in Definition 3. Chapter 9 develops compilation and verification techniques to ensure that the enclave program satisfies the page access obliviousness property that we define in Definition 4.

# Chapter 7

# Moat: Verifying Confidentiality of Enclave's Outputs

Even though trusted platforms such as SGX and Sanctum offer isolation from privileged adversaries, an enclave must still rely on the compromised host OS for basic services such as storage and communication. While the platform prevents the OS from directly accessing enclave's memory, an enclave uses the memory it shares with the untrusted host application to interact with the OS, and this form of interaction gives the attacker a means to control the enclave's inputs and outputs and potentially trigger any vulnerability in the enclave's code. The developer must write safe enclave programs by using their trusted primitives correctly, using safe cryptographic protocols, avoiding traditional bugs due to memory safety violations, etc. For instance, the enclave may suffer from exploits like Heartbleed [47] by using vulnerable SSL implementations, and these exploits have been shown to leak secret cryptographic keys from memory. As another example, the enclave developer may forget to perform integrity checks on input buffers originating in untrusted memory, or may compute on secret data directly in untrusted memory. The attacker may exploit such vulnerabilities to extract sensitive data from enclaves. The goal of this chapter is to develop a verification toolchain to guarantee an absence of such information leaks. An enclave developer can use such a toolchain to prove that enclave programs satisfy confidentiality, i.e., there is no execution that leaks a secret to the adversary-visible, non-enclave memory.

Verifying confidentiality involves tracking the flow of secrets within the application's memory, and proving that the adversary does not observe values that depend on secrets. While past research has produced several type systems that verify information flows (e.g. Jif [86], Volpano et al. [121], Balliu et al. [13]), they make a fundamental assumption that the infrastructure (OS/VMM, etc.) on which the code runs is safe, which is unrealistic due to privileged malware attacks. Furthermore, extending traditional type systems to enclave programs is non-trivial because the analysis must faithfully model the semantics of SGX instructions and infer information flows to individual addresses within enclave memory. Therefore, we develop a static verifier called Moat that analyzes the instruction-level behavior of the enclave binary program. Moat employs a flow- and path-sensitive type checking algo-

rithm (based on automated theorem proving using satisfiability modulo theories solving [17]) for automatically verifying whether an enclave program (in the presence of a privileged adversary) provides confidentiality guarantees. For unsafe programs, Moat returns an exploit demonstrating a potential leak of secret to non-enclave memory. By analyzing the binary, we remove the compiler from our trusted computing base, and also avoid making memory safety assumptions that are common in traditional information flow type systems based on high-level languages. Though we study these issues in the context of TAP (which we prove that it simulates Intel SGX and Sanctum), similar issues arise in other architectures based on trusted hardware such as ARM TrustZone [9] and Sancus [90], and our approach is potentially applicable to them as well. The theory we develop with regard to attacker models and our verifier is mostly independent of the specifics of SGX and Sanctum.

In short, the goal of this chapter is to provide a methodology and tool support for the programmer to write safe enclaves. This chapter has the following organization. In Section 7.1, we give present an overview of the challenges addressed by and solutions adopted by Moat, which is described in technical detail in Section 7.2. Section 7.3 evaluates Moat on several case studies. We discuss several works related to Moat in Section 7.4.

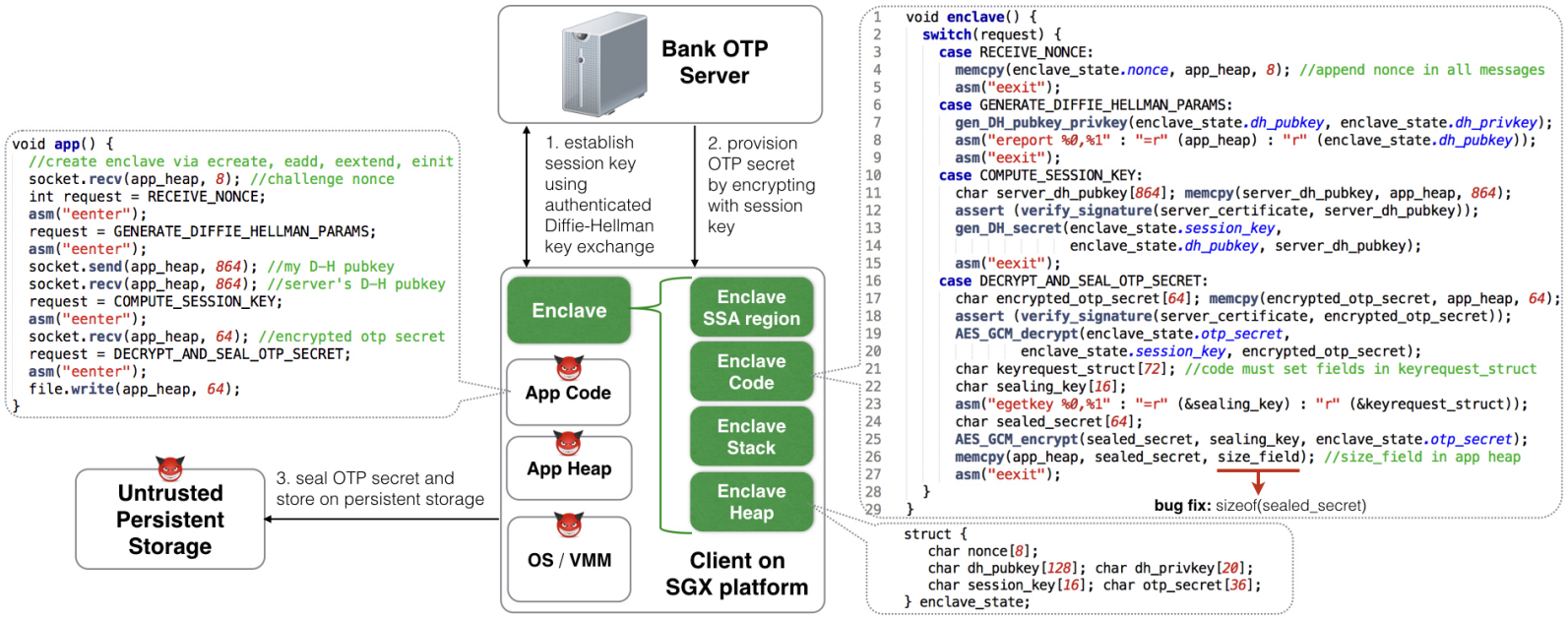
## 7.1 Overview of Moat

In this section, we give an overview of Moat's approach for proving confidentiality properties of enclave code (detailed exposition in Section 7.2). Moat proves that a privileged software adversary (specifically the $M$ adversary defined in Section 4.2; Moat does not defend against side channel leaks) running on the same machine does not observe a value that depends on *Secrets*, regardless of any operations performed by that adversary.

Moat accepts an enclave program in x86 Assembly, containing TAP's enclave-mode instructions `attest`, `rand`, and `exit`. Moat is also given a set of annotations, called *Secrets*, indicating 1) code locations where secret values are generated (e.g. after key generation, decryption, etc.), and 2) memory locations where those secret values are stored. In the OTP application presented in Section 2.3.1 (and repeated here in Figure 7.1), the *Secrets* include otp_secret, session_key, and sealing_key.

We demonstrate Moat's proof methodology on a snippet of OTP enclave code containing lines 22-26 from Figure 7.1, which is first compiled using an off-the-shelf C compiler to x86+SGX Assembly in Figure 7.2. Here, the enclave invokes `egetkey` to retrieve a 128-bit (secret) sealing key, which is stored in the byte array sealing_key. Next, the enclave encrypts otp_secret (using AES-GCM-128 encryption library function called encrypt) to compute the sealed_secret. Finally, the enclave copies sealed_secret to untrusted memory app_heap (to be written to disk). Observe that the size argument to memcpy (line 26 in Figure 2.2) is a variable size_field which resides in non-enclave memory. The adversary can overwrite size_field with an arbitrary value, thus triggering the buffer overrun vulnerability and causing the enclave to leak secrets from its memory.

To reason about enclave code and find such vulnerabilities, Moat first extracts a model




```
void app() {
  //create enclave via ecreate, eadd, eextend, einit
  socket.recv(app_heap, 8); //challenge nonce
  int request = RECEIVE_NONCE;
  asm("eenter");
  request = GENERATE_DIFFIE_HELLMAN_PARAMS;
  asm("eenter");
  socket.send(app_heap, 864); //my D-H pubkey
  socket.recv(app_heap, 864); //server's D-H pubkey
  request = COMPUTE_SESSION_KEY;
  asm("eenter");
  socket.recv(app_heap, 64); //encrypted otp secret
  request = DECRYPT_AND_SEAL_OTP_SECRET;
  asm("eenter");
  file.write(app_heap, 64);
}
```

```
1  void enclave() {
2    switch(request) {
3      case RECEIVE_NONCE:
4        memcpy(enclave_state.nonce, app_heap, 8); //append nonce in all messages
5        asm("eexit");
6      case GENERATE_DIFFIE_HELLMAN_PARAMS:
7        gen_DH_pubkey_privkey(enclave_state.dh_pubkey, enclave_state.dh_privkey);
8        asm("ereport %0,%1" : "=r" (app_heap) : "r" (enclave_state.dh_pubkey));
9        asm("eexit");
10     case COMPUTE_SESSION_KEY:
11       char server_dh_pubkey[864]; memcpy(server_dh_pubkey, app_heap, 864);
12       assert (verify_signature(server_certificate, server_dh_pubkey));
13       gen_DH_secret(enclave_state.session_key,
14                     enclave_state.dh_pubkey, server_dh_pubkey);
15       asm("eexit");
16     case DECRYPT_AND_SEAL_OTP_SECRET:
17       char encrypted_otp_secret[64]; memcpy(encrypted_otp_secret, app_heap, 64);
18       assert (verify_signature(server_certificate, encrypted_otp_secret));
19       AES_GCM_decrypt(enclave_state.otp_secret,
20             enclave_state.session_key, encrypted_otp_secret);
21       char keyrequest_struct[72]; //code must set fields in keyrequest_struct
22       char sealing_key[16];
23       asm("egetkey %0,%1" : "=r" (&sealing_key) : "r" (&keyrequest_struct));
24       char sealed_secret[64];
25       AES_GCM_encrypt(sealed_secret, sealing_key, enclave_state.otp_secret);
26       memcpy(app_heap, sealed_secret, size_field); //size_field in app heap
27       asm("eexit");
28   }
29 }
```

```
struct {
   char nonce[8];
   char dh_pubkey[128]; char dh_privkey[20];
   char session_key[16]; char otp_secret[36];
} enclave_state;
```

*Figure 7.1*: OTP Enclave Program. The enclave performs trusted cryptographic operations, and the host application performs untrusted tasks such as UI and I/O. Note that the enclave invokes SGX instructions, which have corresponding TAP primitives and Sanctum instructions.

```
egetkey                          mem := egetkey(mem, ebx, ecx);
movl $0x8080AC,0x8(%esp)         mem := store(mem,add(esp,8),8080AC);
lea  -0x6e0(%ebp),%eax           eax := sub(ebp, 6e0);
mov  %eax,0x4(%esp)              mem := store(mem,add(esp,4),eax);
lea  -0x720(%ebp),%eax           eax := sub(ebp, 720);
mov  %eax,(%esp)                 mem := store(mem,esp,eax);
call <AES_GCM_encrypt>           mem := AES_GCM_encrypt(mem, esp);
mov  0x700048,%eax               eax := load(mem,700048);
movl %eax,0x8(%esp)              mem := store(mem,add(esp,8),eax);
lea  -0x720(%ebp),%eax           eax := sub(ebp, 720);
mov  %eax,0x4(%esp)              mem := store(mem,add(esp,4),eax);
movl $0x701000,(%esp)            mem := store(mem,esp,701000);
call 802080 <memcpy>             mem := memcpy(mem, esp);
```

*Figure 7.2*: OTP enclave snippet (left) and its formal model $p$ (right)

in the language that we presented in Section 3.5. We denote this model as $p$, which we define in the language presented in Section 3.5 to be a sequence of TAP instructions. Recall from Section 3.5 that an enclave program is modeled as a sequence of instructions ($Instr$) with a unique entrypoint — since we are only demonstrating Moat on a tiny snippet of enclave code, assume that the execution starts at the `egetkey` instruction. Figure 7.2 shows the model $p$ of the snippet of OTP enclave. Moat leverages the formal semantics of enclave instructions that we developed in Section 3.5 to reason about the enclave's behaviors.

The model extraction is built upon the BAP framework [32], which lifts popular instruction sets (including x86 and ARM) to a simpler intermediate language that resembles our language from Section 3.5. BAP models user-mode x86 instructions precisely, including updates to ISA-defined state such as registers and flags. For brevity, our model in Figure 7.2 omits all updates to flags as they are irrelevant to this code snippet. To avoid reasoning about the probabilistic semantics of cryptographic operators, Moat does not model the implementation of cryptographic routines (such as AES_GCM_encrypt in Figure 7.2). Instead, it replaces all calls to the cryptographic routines with their specifications. For instance, in the case of AES_GCM_encrypt, the specification states that the output ciphertext is not secret (i.e., it can be safely output to non-enclave memory), and that it ensures memory safety: only the allocated buffer for the ciphertext is modified by the call.

Since the enclave program executes in the presence of an active adversary, specifically $M$, we must model the effects of $M$'s operations on the enclave's execution. We follow the strategy discussed in Section 6.1. The integrity theorem of secure remote execution (from Section 5.1) states that any sequence of operations of a privileged adversary can be simulated by an adversary that only modifies all locations in non-enclave memory. Effectively, the adversary is supplying a fresh, arbitrarily-chosen value of non-enclave memory at each step of enclave execution — further recall from Section 3.6 that we limit the attacker's capability to only supplying inputs via untrusted memory, before each step of enclave's execution. To model the adversary's effect on $p$'s execution, we introduce a "havoc $\texttt{mem}_{\neg\texttt{evrange}}$" operation before each enclave instruction in our model, as seen in Figure 7.3. Here, $\texttt{mem}_{\texttt{evrange}}$

denotes enclave's protected virtual address space, and $\texttt{mem}_{\neg\texttt{evrange}}$ is all of untrusted, non-enclave memory; havoc $\texttt{mem}_{\neg\texttt{evrange}}$ updates each address in $\texttt{mem}_{\neg\texttt{evrange}}$ with an arbitrarily chosen value. According to our integrity proof in Section 5.1, this method of modeling $M$ accounts for all privileged adversarial operations — any property we prove on the instrumented model will hold at runtime when the enclave executes in the presence of a privileged adversary. Given an enclave program model $p$, we denote the instrumented model as $p_M$.

```
1  havoc mem¬evrange; mem := egetkey(mem, ebx, ecx);
2  havoc mem¬evrange; mem := store(mem,add(esp,8),8080AC);
3  havoc mem¬evrange; eax := sub(ebp, 6e0);
4  havoc mem¬evrange; mem := store(mem,add(esp,4),eax);
5  havoc mem¬evrange; eax := sub(ebp, 720);
6  havoc mem¬evrange; mem := store(mem,esp,eax);
7  havoc mem¬evrange; mem := AES_GCM_encrypt(mem, esp);
8  havoc mem¬evrange; eax := load(mem,700048);
9  havoc mem¬evrange; mem := store(mem,add(esp,8),eax);
10 havoc mem¬evrange; eax := sub(ebp, 720);
11 havoc mem¬evrange; mem := store(mem,add(esp,4),eax);
12 havoc mem¬evrange; mem := store(mem,esp,701000);
13 havoc mem¬evrange; mem := memcpy(mem, esp);
```

*Figure 7.3*: Enclave Model $p$ instrumented with Adversary $M$'s operations, denoted $p_M$.

As we discussed in Section 2.3.1, the OTP enclave implementation is vulnerable. The size argument to memcpy (line 26 in Figure 7.1) is a field within a data structure in non-enclave memory. This vulnerability manifests within $p_M$ as a `load` (line 8 of Figure 7.3), which reads a value from non-enclave memory and passes that value as the size argument to memcpy. To perform the exploit, the attacker $M$ uses havoc $\texttt{mem}_{\neg\texttt{evrange}}$ (in line 8) to control the number of bytes that the enclave writes to non-enclave memory, starting at the base address of sealed_secret. By setting this value to be greater than the size of sealed_secret, $M$ causes the enclave to leak the stack contents, which includes the sealing_key. Note that the specification of AES_GCM_encrypt allows us to safely write out sealed_secret to non-enclave memory because it is encrypted, but writing out other parts of memory (e.g. sealing_key) is unsafe. We formalize a confidentiality property in Chapter 6 that captures such vulnerabilities, and build a static type system in Section 7.2 which only admits programs that satisfy confidentiality. Confidentiality enforces that for any pair of traces of the instrumented model that differ in the values of *Secrets*, if $M$'s operations along the two traces are equivalent (i.e., equivalent sequences of non-enclave memories), then $M$'s observations along the two traces must also be equivalent. In other words, $M$'s observation of enclave's execution is independent of *Secrets*. Note that side channels are out of scope, which is why we choose the threat model of Moat to be the adversary $M$.

Our type system checks confidentiality by further instrumenting the enclave model with ghost variables that track the flow of *Secrets* within registers and memory, akin to taint

```
1  assert ¬C_ecx; C_mem^old := C_mem; havoc C_mem;
2  assume ∀i. (ecx ≤ i < ecx + 16) → C_mem[i] ↔ true;
3  assume ∀i. ¬(ecx ≤ i < ecx + 16) → C_mem[i] ↔ C_mem^old[i];
4  havoc mem_¬evrange; mem := egetkey(mem, ebx, ecx);
5  assert ¬C_esp; C_mem[add(esp, 8)] := false;
6  havoc mem_¬evrange; mem := store(mem,add(esp,8),8080AC);
7  C_eax := C_ebp;
8  havoc mem_¬evrange; eax := sub(ebp, 6e0);
9  assert ¬C_esp ∧ (¬evrange(add(esp, 4)) → ¬C_eax); C_mem[add(esp, 4)] := C_eax;
10 havoc mem_¬evrange; mem := store(mem,add(esp,4),eax);
11 C_eax := C_ebp;
12 havoc mem_¬evrange; eax := sub(ebp, 720);
13 assert ¬C_esp ∧ (¬evrange(esp) → ¬C_eax); C_mem[esp] := C_eax;
14 havoc mem_¬evrange; mem := store(mem, esp, eax);
15 C_mem := C_AES_GCM_encrypt(C_mem, esp);
16 havoc mem_¬evrange; mem := AES_GCM_encrypt(mem, esp);
17 C_eax := C_mem[700048];
18 havoc mem_¬evrange; eax := load(mem, 700048);
19 assert ¬C_esp ∧ (¬evrange(add(esp, 8)) → ¬C_eax); C_mem[add(esp, 8)] := C_eax;
20 havoc mem_¬evrange; mem := store(mem, add(esp,8), eax);
21 C_eax := C_ebp;
22 havoc mem_¬evrange; eax := sub(ebp, 720);
23 assert ¬C_esp ∧ (¬evrange(add(esp, 4)) → ¬C_eax); C_mem[add(esp, 4)] := C_eax;
24 havoc mem_¬evrange; mem := store(mem, add(esp,4), eax);
25 assert ¬C_esp; C_mem[esp] := false;
26 havoc mem_¬evrange; mem := store(mem, esp, 7001000);
27 C_mem := C_memcpy(C_mem, esp);
28 arg1 := load(mem, esp); arg3 := load(mem, add(esp, 8));
29 havoc mem_¬evrange; mem := memcpy(mem, esp);
30 assert ∀i. ((arg1 ≤ i < add(arg1, arg3)) ∧ ¬enc(i)) → ¬C_mem[i];
```

*Figure 7.4*: $I(p_M)$: enclave Model instrumented with ghost variables for tracking flow of secrets, and assertions for checking safety of information flows.

tracking but performed using static analysis, and assertions which check for valid information flows. Figure 7.4 illustrates this transformation for the enclave snippet from Figure 7.3. Given an enclave model $p_M$, which contains the attacker's havoc $\texttt{mem}_{\neg\texttt{evrange}}$ operations, we denote the instrumented model (which contains ghost variables and typing assertions) as $I(p_m)$. Moat tracks both implicit and explicit information flows [100] using the instrumentation in $I(p_M)$. For each state variable x, the type system instruments a ghost variable $\mathsf{C_x}$. $\mathsf{C_x}$ is updated on each assignment that updates x, and is assigned to *false* only if x's value is *not* a function of *Secrets* (details in Section 7.2). For instance, $\mathsf{C_{mem}}$[esp] in line 13 is assigned to $\mathsf{C_{eax}}$ because a secret in the eax register makes the written memory location also secret. Furthermore, for each secret in *Secrets*, we set the corresponding locations in $\mathsf{C_{mem}}$ to *true*. For instance, lines 1-3 assign *true* to those 16 bytes in $\mathsf{C_{mem}}$ where `egetkey` places the secret sealing_key. Information leaks can only happen via `store` to $\texttt{mem}_{\neg\texttt{evrange}}$,

since the adversary cannot observe $\texttt{mem}_{\texttt{evrange}}$. For each `store` instruction, the type system instruments an `assert` checking that a secret value is not written to $\texttt{mem}_{\neg\texttt{evrange}}$ (with special treatment of memcpy for efficiency) — recall that the predicate $\texttt{evrange}(i)$ is *true* if i is an address in $\texttt{mem}_{\texttt{evrange}}$. For a program to be well-typed, all assertions in the instrumented instrumented model must be valid along any feasible execution.

Moat feeds the instrumented program (Figure 7.4) to a static program verifier, which uses SMT solving to explore all executions (i.e., all reachable states) and verify that the assertions are valid along all executions. The assertion in line 30 is invalid because $\mathsf{C}_{\mathsf{mem}}$ is *true* for memory locations that hold the sealing_key. Our type system rejects this enclave program. A fix to the OTP implementation is to replace size_field with the correct size, which is 64 bytes. Although memory safety vulnerabilities can be found using simpler static analysis, Moat can identify several classes of vulnerabilities using these typing assertions, at the level of machine code.

### 7.1.1 Declassification

In the previous section, we claim that writing sealed_secret to $\texttt{mem}_{\neg\texttt{evrange}}$ is safe because it is encrypted using a secret key. We now explain how Moat evaluates whether a particular enclave output is safe. As a pragmatic choice, Moat does not reason about cryptographic operations for there is significant body of research on cryptographic protocol verification. For instance, if encryption uses a key established by Diffie-Hellman, the verification would need to reason about the authentication and attestation scheme used in that Diffie-Hellman exchange in order to derive that the key can be safely used for encryption. Protocol verifiers (e.g. ProVerif [25], CryptoVerif [26]) excel at this form of reasoning. Therefore, when Moat encounters a cryptographic library call, it abstracts it as an uninterpreted function with the conservative axiom that secret inputs produce secret output.

However, this conservative axiomatization is unnecessary because a secret encrypted with a key (that is unknown to the adversary) can be safely output. A declassified output is a intentional information leak of the program, which may be proven to be a safe using other proof techniques. In our experiments, we safely eliminate declassified outputs from information leakage checking if the protocol verifier has already proven them to be safe outputs. For instance, since the AES-GCM cipher can be safely assumed to provide semantic security, in Figure 7.4, AES_GCM_encrypt on line 16 is an uninterpreted function, and C_AES_GCM_encrypt on line 15 marks the ciphertext as non-secret (i.e., public).

To collect the $Declassified$ annotations, we manually model the cryptographic protocol to verify using an off-the-shelf protocol verifier. The choice of protocol verifier is orthogonal to our work. A protocol verifier accepts as input an abstract model of the protocol (in a formalism such as spi calculus [5]), and proves properties such as confidentiality of protocol-level secrets. We briefly describe how we use Moat in tandem with a protocol verifier. If Moat establishes that a particular value generated by the enclave is secret, this can be added to the set of secrecy assumptions made in the protocol verifier. Similarly, if the protocol verifier establishes confidentiality even while assuming that an enclave's output is observable by the

adversary, then we can declassify that output while verifying the enclave with Moat. This assume-guarantee reasoning is sound because the adversary model used by Moat (i.e., $M$) can simulate a network adversary — a network adversary reorders, inserts, and deletes messages, and the observable effect of these operations can be simulated by a havoc $\texttt{mem}_{\neg\texttt{evrange}}$.

We demonstrate this assume-guarantee reasoning on lines 22-26 of the OTP enclave in Figure 7.4, where line 26 no longer has the memory safety vulnerability, i.e., it uses the constant 64 instead of size_field. Despite the fix, Moat is initially unable to prove that memcpy in line 26 of Figure 7.4 is safe because its axiomatization of aes_gcm_encrypt is imprecise. We proceed by first proving in Moat that the sealing_key (obtained using egetkey) is not leaked to the adversary. Next, we annotate the ProVerif model with the assumption that sealing_key is secret, which allows ProVerif to prove that the outbound message (via memcpy) is safe. Based on this ProVerif proof, we annotate the sealed_secret as $Declassified$, hence telling Moat that the assert on line 30 of Figure 7.4 is valid.

This illustrates that protocol verification not only provides $Declassified$ annotations, but also specifies which values must be kept secret by the enclave to ensure that the protocol is safe. The combination of $Secrets$ and $Declassified$ annotations is called a policy, and this policy forms an input to Moat in addition to the enclave program.

### 7.1.2 Assumptions and Limitations

Moat has the following fundamental limitations:

- To make static analysis feasible, Moat assumes that the enclave code cannot be modified at runtime (enforced using permissions on enclave's pages).
- Moat assumes that the enclave program has control flow integrity. Moat does not find vulnerabilities that exploit the control flow behavior (such as ROP attacks). This assumption is not fundamental, and can be removed using a combination of modern runtime defenses (e.g. [6]) and static analysis tools [111]). In fact, we develop a compiler and verifier in Chapter 8 that guards against such attacks while establishing confidentiality of the enclave program.
- We do not consider attacks from observing side channels such as memory access patterns, etc. We address this side channel in Chapter 9. We also do not address timing channels, and the enclave program can leak secret data via (the time of ) termination. The confidentiality guarantee checked by Moat is a form of termination-insensitive non-interference [64].
- Moat's type system prevents programs from accessing memory using secret-dependent addresses, for simplicity. This may be restrictive in practice, and we find that Moat is typically suited for programs where secrets consist only of cryptographic keys, and their use is confined to cryptographic operations. We do relax this restriction in Chapter 8.

- Although SGX allows an enclave to have multiple CPU threads, we only consider single-threaded enclaves for simplicity.

The current implementation of Moat makes the following additional assumptions:

- The TAP platform, specifically the SGX and Sanctum processor, is in our trusted computing base. Specifically, we assume that BAP models all machine instructions faithfully, and that the concrete processor hardware fulfills the ISA-defined semantics. This eliminates a class of attacks, such as physical tampering of the CPU and supply chain attacks, from the threat model.

- Moat's implementation uses the Boogie [14] program verifier, Z3 [41] SMT solver, and BAP [32] for modeling x86 instructions. All these dependencies are in our trusted computing base.

- We use trusted implementation of cryptographic routines (cryptopp library [1]) to develop our benchmarks. Since Moat does not model their implementation, they are in our trusted computing base.

- We assume that the enclave code does not handle exceptions, apart from page fault exceptions which are handled seamlessly by the OS/VMM. In other words, we terminate the enclave in the event of all other exceptions (such as divide by 0).

- Moat assumes that the enclave code does not read (via `load` instruction) from static save area (SSA). We have not yet found this to be a limiting assumption in any practical program, including our benchmarks. Note that this assumption does not prevent the untrusted code from invoking `eresume` (which is necessary for resuming from asynchronous exits).

- Moat performs a global analysis of the enclave code, and it must inline all procedure calls to that end. While this simplifies the low-level assertions that we need to prove, it limits scalability.

## 7.2 Proving Confidentiality

Moat automatically checks if the model of enclave program satisfies confidentiality (property 6.1). Since confidentiality is a 2-safety hyperproperty (property over pairs of traces), we cannot use black box program verification techniques, which are tailored towards safety properties. Hence, we create a security type system in which type safety implies that the enclave satisfies confidentiality. We avoid a self-composition approach because of complications in encoding equivalence assumptions over adversary operations in the two traces of the enclave (property 6.1). As is standard in many type-based systems [121, 86], the typing rules prevent programs that have explicit and implicit information leaks. Explicit leaks occur via assignments of secret values to $M$-observable state, i.e., $\texttt{mem}_{\neg\texttt{evrange}}$. For instance,

mem := store(mem,a,d) is ill-typed if d's value depends on a secret and evrange(a) is false, i.e., it writes a secret to non-enclave memory. An implicit leak occurs when a conditional statement has a secret-dependent guard, but updates $M$-visible state in either branch. For instance, if (d == 42) {mem := store(mem, a, 1)} else {skip} is ill-typed if d's value depends on a secret and evrange(a) is false. In both examples above, $M$ learns the secret value d by reading mem at address a. Moat's type system precisely models the semantics of x86 instructions and TAP primitives, and certifies the enclave only if it has no implicit or explicit leaks. However, as we mention before, Moat's type system prevents programs from accessing memory using secret-dependent addresses. This may be restrictive in practice, and we find that Moat is typically suited for programs where secrets consist only of cryptographic keys, and their use is confined to cryptographic operations.

We now describe the specifics of Moat's type system. A security type is either $\top$ (secret) or $\bot$ (public). At each program point, each memory location and CPU register has a security type based on the machine instructions executed until that location. The security types are needed at each program location because variables (especially regs) may alternate between holding secret and public values. As explained later in this section, Moat uses the security types in order to decide whether any computation in the enclave can produce implicit or explicit leaks. We assume that the developer has provided the $Secrets = \{(l, v)\}$ and $Declassified = \{(l, v)\}$ annotations. We don't require the developer to provide any other annotations; therefore, Moat implements a type inference algorithm based on computing refinement type constraints and checking their validity using a theorem prover. In contrast, type checking without inference would require the programmer to painstakingly provide security types for each memory location and CPU register, at each program point — flow sensitivity and type inference are key requirements of type checking machine code.

Moat's type inference algorithm computes first-order logical constraints under which an expression or statement takes a security type. A typing judgment $\vdash e : \tau \Rightarrow \psi$ means that the expression $e$ has security type $\tau$ whenever the constraint $\psi$ is satisfied. An expression of the form $op(v_1, \ldots, v_n)$ (where $op$ is a relation or function) has type $\tau$ if all variables $\{v_1, \ldots, v_n\}$ have type $\tau$ or lower. That is, an expression has type $\bot$ iff its value is independent of $Secrets$.

For an enclave program $p$ to have type $\tau$, every assignment in $p$ must update a state variable whose security class is $\tau$ or higher. We write this typing judgment as $[\tau] \vdash p \Rightarrow \langle \psi, \mathcal{F} \rangle$, where $\psi$ is a first-order (SMT) formula and $\mathcal{F}$ is a set of first-order (SMT) formulae. Each satisfiable interpretation of $\psi$ corresponds to a feasible execution of $p$. $\mathcal{F}$ contains a SMT formula for each instruction $i$ in $p$ (recall that $p$ is a sequence of instructions), such that the formula is valid iff that instruction does not leak secrets. We present our typing rules in Figure 7.5, which assume that the enclave model is first converted to single static assignment form. An instruction $i$ has type $\tau$ if we derive $[\tau] \vdash i \Rightarrow \langle \psi, \mathcal{F} \rangle$ using the typing rules, and prove (using a First-Order Logic theorem prover) that all formulae in $\mathcal{F}$ are valid. If $i$ has type $\top$, then $i$ does not update $M$-visible state, and thus cannot contain information leaks. Having type $\top$ also allows $i$ to execute in a context where a secret value is implicitly known through the guard of a conditional statement. On the other hand, type $\bot$ implies that $i$ either does not update $M$-observable state or the update is independent of $Secrets$.

We now explain some of our typing rules from Figure 7.5. For each variable $\mathsf{v} \in \mathit{Vars}$, our typing rules introduce a ghost variable $\mathsf{C_v}$ that is *true* iff $\mathsf{v}$ has security type $\top$. For a scalar register $\mathsf{v}$, $\mathsf{C_v}$ is a boolean; for an array variable $\mathsf{v}$, $\mathsf{C_v}$ (e.g. $\mathsf{C_{mem}}$) is an array and $\mathsf{C_v}[\mathsf{i}]$ denotes the security type for each location $i$. The *exp1* rule allows inferring the type of any expression $e$ as $\top$. The *exp2* rule allows inferring an expression type $e$ as $\bot$ if we derive $\mathsf{C_v}$ to be false for all variables $\mathsf{v}$ in the expression $e$. The *storeL* rule marks the memory location as secret if the stored data is secret. In case of secret data, we assert that the updated location is within $\mathtt{mem}_{\neg\mathtt{evrange}}$; we also assert that the address is public to prevent implicit leaks. Since the *storeH* rule types `store` instructions as $\top$, it unconditionally marks the memory location as secret. This is necessary because the `store` may execute in a context where a secret is implicitly known through the guard of a conditional statement. The *load* rule marks the updated register as secret if the memory location contains a secret value. The *attestL* rule types the updated memory locations as per SGX semantics. `attest` takes 64 bytes of data (that the programmer intends to bind to the measurement) at address in ecx, and copies them to memory starting at address $\mathsf{edx} + 320$; the rest of the report has public data such as the MAC, measurement, etc. Hence, $\mathsf{C_{mem}}$ retains the secrecy level for the 64 bytes of data, and assumes $\mathit{false}$ for the public data. Similar to *storeH*, *attestH* unconditionally marks all 432 bytes of the report as secret. Notice that we do not assert that `attest` writes to enclave memory since this is checked by the TAP. `exit` jumps to the host application without clearing `regs`. Hence, the *exit* rule asserts that those `regs` hold public values.

Observe that we do not include any rules for procedure calls and returns. This is because Moat inlines all procedure calls in the enclave's main method, and performs a global analysis of the resulting program. While this certainly limits the scalability of Moat, it allows our typing rules to be simpler. We present an alternative technique for verifying enclaves in Chapter 8 which relaxes this limitation.

**Theorem 2** *For any enclave program $p$ such that $[\tau] \vdash p \Rightarrow (\psi, \mathcal{F})$ is derivable (where $\tau$ is either $\top$ or $\bot$) and all formulae in $\mathcal{F}$ are valid, $p$ satisfies property 6.1.*

Moat implements this type system by translating each instruction $i$ in $p_M$ to $I(i)$ using the rules in Figure 7.6. Observe that we introduce $\mathsf{C_{pc}}$ to track whether confidential information is implicitly known through the program counter. If a conditional statement's guard depends on a secret value, then we set $\mathsf{C_{pc}}$ to *true* within the then and else branches. Moat invokes $I(p_M)$ and applies the instrumentation rules in Figure 7.6 recursively. Figure 7.4 demonstrates an example of instrumenting $p_M$. Moat then feeds the instrumented program $I(p_M)$ to an off-the-shelf program verifier, which explores all feasible executions, and proves validity all assertions or finds a counter-example. Our implementation uses the Boogie [14] program verifier, which receives $I(p_M)$ and generates verification conditions in the SMT format. Boogie uses the Z3 [41] theorem prover (SMT solver) to prove the verification conditions. An advantage of using SMT solving is that a typing error is explained using counter-example execution, demonstrating the information leak and exploit.

$$\frac{}{\vdash e : \top \Rightarrow true}(exp1) \qquad \frac{[\top] \vdash i \Rightarrow \langle \psi, A \rangle}{[\bot] \vdash i \Rightarrow \langle \psi, A \rangle}(coercion) \qquad \frac{}{[\tau] \vdash \texttt{assume}\ \phi \Rightarrow \langle \phi, \{\emptyset\} \rangle}(assume)$$

$$\frac{}{\vdash e : \bot \Rightarrow \bigwedge_{\mathsf{v} \in Vars(e)} \neg \mathsf{C_v}}(exp2) \qquad \frac{}{[\top] \vdash \mathsf{skip} \Rightarrow \langle true, \{\emptyset\} \rangle}(skip) \qquad \frac{}{[\tau] \vdash \texttt{assert}\ \phi \Rightarrow \langle \phi, \{\phi\} \rangle}(assert)$$

$$\frac{}{[\tau] \vdash \mathsf{x}' := e \Rightarrow \langle (\mathsf{x}' = e) \wedge (\mathsf{C_{x'}} \leftrightarrow \bigvee_{\mathsf{v} \in Vars(e)} \mathsf{C_v}), \{\emptyset\} \rangle}(scalar)$$

$$\frac{\vdash e_a : \bot \Rightarrow \psi_a}{[\tau] \vdash \mathsf{x}' := \texttt{load}(\mathtt{mem}, e_a) \Rightarrow \langle (\mathsf{x}' = \texttt{load}(\mathtt{mem}, e_a)) \wedge (\mathsf{C_{x'}} \leftrightarrow \mathsf{C_{mem}}[e_a]), \{\psi_a\} \rangle}(load)$$

$$\frac{\vdash e_a : \bot \Rightarrow \psi_a}{\begin{array}{l}[\top] \vdash \mathtt{mem}' := \texttt{store}(\mathtt{mem}, e_a, e_d) \Rightarrow \\ \langle \mathtt{mem}' = \texttt{store}(\mathtt{mem}, e_a, e_d) \ \wedge \ \mathsf{C_{mem'}} = \mathsf{C_{mem}}[e_a := true], \{\texttt{evrange}(\mathsf{e_a}) \wedge \psi_a\} \rangle\end{array}}(storeH)$$

$$\frac{\vdash e_a : \bot \Rightarrow \psi_a \qquad \vdash e_d : \bot \Rightarrow \psi_d}{\begin{array}{l}[\bot] \vdash \mathtt{mem}' := \texttt{store}(\mathtt{mem}, e_a, e_d) \Rightarrow \\ \langle \mathtt{mem}' = \texttt{store}(\mathtt{mem}, e_a, e_d) \wedge \mathsf{C_{mem'}} = \mathsf{C_{mem}}[e_a := \psi_d], \{\psi_a \wedge (\neg \texttt{evrange}(e_a) \rightarrow \psi_d)\} \rangle\end{array}}(storeL)$$

$$\frac{}{\begin{array}{l}[\bot] \vdash \mathtt{mem}' := \texttt{attest}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx}) \Rightarrow \\ \langle \mathtt{mem}' = \texttt{ereport}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx}) \wedge \\ \forall j.\ (\mathsf{edx} \leq j < \mathsf{edx} + 320) \rightarrow \neg \mathsf{C_{mem'}}[j] \\ \wedge\ (\mathsf{edx} + 320 \leq j < \mathsf{edx} + 384) \rightarrow \mathsf{C_{mem'}}[j] \leftrightarrow \mathsf{C_{mem}}[\mathsf{ecx} + j - \mathsf{edx} - 320] \\ \wedge\ (\mathsf{edx} + 384 \leq j < \mathsf{edx} + 432) \rightarrow \mathsf{C_{mem'}}[j] \leftrightarrow false \\ \wedge\ \neg(\mathsf{edx} \leq j < \mathsf{edx} + 432) \rightarrow \mathsf{C_{mem'}}[j] \leftrightarrow \mathsf{C_{mem}}[j], \{\neg \mathsf{C_{edx}}\} \rangle\end{array}}(attestL)$$

$$\frac{}{\begin{array}{l}[\top] \vdash \mathtt{mem}' := \texttt{attest}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx}) \Rightarrow \\ \langle \mathtt{mem}' = \texttt{ereport}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx}) \wedge \\ \forall j.\ (\mathsf{edx} \leq j < \mathsf{edx} + 432) \rightarrow (\mathsf{C_{mem'}}[j] \leftrightarrow true) \\ \wedge\ \neg(\mathsf{edx} \leq j < \mathsf{edx} + 432) \rightarrow (\mathsf{C_{mem'}}[j] \leftrightarrow \mathsf{C_{mem}}[j]), \{\neg \mathsf{C_{edx}}\} \rangle\end{array}}(attestH)$$

$$\frac{}{[\tau] \vdash \mathtt{mem}', \mathtt{regs}' := \texttt{eexit}(\mathtt{mem}) \Rightarrow \langle (\mathtt{mem}', \mathtt{regs}') = \texttt{eexit}(\mathtt{mem}), \{\forall \mathsf{r} \in \mathtt{regs}.\ \neg \mathsf{C_r}\} \rangle}(exit)$$

$$\frac{[\tau] \vdash i_1 \Rightarrow \langle \psi_1, \mathcal{F}_1 \rangle \qquad [\tau] \vdash i_2 \Rightarrow \langle \psi_2, \mathcal{F}_2 \rangle}{[\tau] \vdash i_1; i_2 \Rightarrow \langle \psi_1 \wedge \psi_2, \mathcal{F}_1 \cup \{\psi_1 \rightarrow f_2 \mid f_2 \in \mathcal{F}_2\} \rangle}(seq)$$

$$\frac{\vdash e : \tau \Rightarrow \psi \quad [\tau] \vdash i_1 \Rightarrow \langle \psi_1, \mathcal{F}_1 \rangle \quad [\tau] \vdash i_2 \Rightarrow \langle \psi_2, \mathcal{F}_2 \rangle}{[\tau] \vdash \mathsf{if}\ (e)\ \{i_1\}\ \mathsf{else}\ \{i_2\} \Rightarrow \langle (e \rightarrow \psi_1) \wedge (\neg e \rightarrow \psi_2), \{\psi\} \cup \{e \rightarrow f_1 \mid f_1 \in \mathcal{F}_1\} \cup \{\neg e \rightarrow f_2 \mid f_2 \in \mathcal{F}_2\} \rangle}(ite)$$

*Figure 7.5*: Typing Rules for $p_M$

| Instruction $i$ | Instrumented Instruction $I(i)$ |
|---|---|
| $\texttt{assert}\ \phi$ | $\texttt{assert}\ \phi$ |
| $\texttt{assume}\ \phi$ | $\texttt{assume}\ \phi$ |
| skip | skip |
| havoc $\texttt{mem}_{\neg\mathsf{epc}}$ | havoc $\texttt{mem}_{\neg\mathsf{epc}}$ |
| $\mathsf{x} := e$ | $\mathsf{C_x} := \mathsf{C_{pc}} \vee \bigvee_{v \in Vars(e)} \mathsf{C_v}$; $\mathsf{x} := e$ |
| $\mathsf{x} := \texttt{load}(\texttt{mem}, e)$ | $\texttt{assert}\ \bigwedge_{v \in Vars(e)} \neg\mathsf{C_v}$;<br>$\mathsf{C_x} := \mathsf{C_{pc}} \vee \mathsf{C_{mem}}[e]$; $\mathsf{x} := \texttt{load}(\texttt{mem}, e)$ |
| $\texttt{mem} :=$<br>$\texttt{store}$<br>$(\texttt{mem}, e_a, e_d)$ | $\texttt{assert}\ \bigwedge_{v \in Vars(e_a)} \neg\mathsf{C_v}$; $\texttt{assert}\ \mathsf{C_{pc}} \rightarrow \texttt{evrange}(e_a)$;<br>$\texttt{assert}\ (\neg\mathsf{C_{pc}} \wedge \neg\texttt{evrange}(e_a)) \rightarrow (\bigwedge_{v \in Vars(e_d)} \neg\mathsf{C_v})$;<br>$\mathsf{C_{mem}}[\mathsf{e_a}] := \mathsf{C_{pc}} \vee \bigvee_{v \in Vars(e_d)} \mathsf{C_v}$;<br>$\texttt{mem} := \texttt{store}(\texttt{mem}, \mathsf{e_a}, e_d)$ |
| $\texttt{mem} :=$<br>$\texttt{ereport}$<br>$(\texttt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx})$ | $\texttt{assert}\ \neg\mathsf{C_{edx}}$; $\mathsf{C^{old}_{mem}} := \mathsf{C_{mem}}$; havoc $\mathsf{C_{mem}}$;<br>$\texttt{assume}\ \forall j.\ (\mathsf{edx} \leq j < \mathsf{edx} + 320) \rightarrow \mathsf{C_{mem}}[j] = \mathsf{C_{pc}}$;<br>$\texttt{assume}\ \forall j.\ (\mathsf{edx} + 320 \leq j < \mathsf{edx} + 384) \rightarrow$<br>$\quad \mathsf{C_{mem}}[j] = (\mathsf{C_{pc}} \vee \mathsf{C^{old}_{mem}}[\mathsf{ecx} + j - \mathsf{edx} - 320])$;<br>$\texttt{assume}\ \forall j.\ (\mathsf{edx} + 384 \leq j < \mathsf{edx} + 432) \rightarrow$<br>$\quad \mathsf{C_{mem}}[j] = \mathsf{C_{pc}}$;<br>$\texttt{assume}\ \forall j.\ \neg(\mathsf{edx} \leq j < \mathsf{edx} + 432) \rightarrow \mathsf{C_{mem}}[j] = \mathsf{C^{old}_{mem}}[j]$;<br>$\texttt{mem} := \texttt{ereport}(\texttt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx})$ |
| $\texttt{mem} :=$<br>$\texttt{egetkey}$<br>$(\texttt{mem}, \mathsf{ebx}, \mathsf{ecx})$ | $\texttt{assert}\ \neg\mathsf{C_{ecx}}$; $\mathsf{C^{old}_{mem}} := \mathsf{C_{mem}}$; havoc $\mathsf{C_{mem}}$;<br>$\texttt{assume}\ \forall j.\ (\mathsf{ecx} \leq j < \mathsf{ecx} + 16) \rightarrow \mathsf{C_{mem}}[j]$;<br>$\texttt{assume}\ \forall j.\ \neg(\mathsf{ecx} \leq j < \mathsf{ecx} + 16) \rightarrow \mathsf{C_{mem}}[j] = \mathsf{C^{old}_{mem}}[j]$;<br>$\texttt{mem} := \texttt{egetkey}(\texttt{mem}, \mathsf{ebx}, \mathsf{ecx})$ |
| $\texttt{mem}, \texttt{regs} :=$<br>$\texttt{eexit}(\texttt{mem}, \mathsf{ebx})$ | $\texttt{assert}\ \forall \mathsf{r} \in \texttt{regs}.\ \neg\mathsf{C_r}$;<br>$\texttt{mem}, \texttt{regs} := \texttt{eexit}(\texttt{mem})$ |
| $i_1\ ;\ i_2$ | $I(i_1); I(i_2)$ |
| $\mathsf{if}(e)\{i_1\}\mathsf{else}\{i_2\}$ | $\mathsf{C^{in}_{pc}} := \mathsf{C_{pc}}$;<br>$\mathsf{C_{pc}} := \mathsf{C_{pc}} \vee \bigvee_{v \in Vars(e)} \mathsf{C_v}$;<br>$\mathsf{if}\ (e)\ \{I(i_1)\}\ \mathsf{else}\ \{I(i_2)\}$;<br>$\mathsf{C_{pc}} := \mathsf{C^{in}_{pc}}$ |

*Figure 7.6*: Instrumentation rules for $p_M$

## 7.3 Evaluation

Moat's implementation comprises (1) translation from machine code program to $p$ using BAP, (2) transformation to $p_M$ by instrumenting havoc $\texttt{mem}_{\neg\texttt{evrange}}$ operations before each instruction $i$ in $p$, (3) transformation to $I(p_M)$ using the rules in Table 8.1, and (4) invoking Boogie/Z3 [41] Theorem Prover to prove validity of all assertions in $I(p_M)$ (modulo explicit declassifications, provided by the user).

### 7.3.1 Optimizations

Our primary objective with Moat was to build a sound verifier for proving confidentiality in the presence of an active adversary. However, we recognize certain scalability challenges, and implement the following optimizations to help Z3 prove the typing assertions. First, we only introduce havoc $\texttt{mem}_{\neg\texttt{evrange}}$ prior to `load` instructions because only a `load` can be used to read non-enclave memory $\texttt{mem}_{\neg\texttt{evrange}}$, and other instructions are not affected by the contents of non-enclave memory. Furthermore, we axiomatize specific library calls such as memcpy and memset, because their loopy implementations incur significant verification overhead.

### 7.3.2 Case Studies

We now describe some case studies which we verified using Moat and ProVerif in tandem, and summarize the results in Figure 7.7. We use the following standard crytpographic notation and assumptions. $m_1|\ldots|m_n$ denotes concatenation of $n$ messages. We use a keyed-hash message authentication function $\mathsf{MAC}_k(text)$ and hash function $\mathsf{H}(text)$, both of which are assumed to be collision-resistant. For asymmetric cryptography, $K_e^{-1}$ and $K_e$ are principal $e$'s private and public signature keys, where we assume that $K_e$ is long-lived and distributed within certificates signed by a root of trust authority. Digital signature using a key $k$ is written as $\mathsf{Sig}_k(text)$; we assume unforgeability under chosen message attacks. We assume that the processor vendor provisions each TAP processor with a unique private key $K_{TAP}^{-1}$ that is available to a special quoting enclave. Using a protocol with the quoting enclave, an enclave can invoke `attest` to produce quotes, which is essentially a signature (using the private key $K_{SGX}^{-1}$) of the data produced by the enclave and its measurement. We write a quote produced on behalf of enclave $e$ as $\mathsf{Quote}_e(text)$, which is equivalent to $\mathsf{Sig}_{K_{TAP}^{-1}}(\mathsf{H}(text) \mid M_e)$ — measurement of enclave $e$ is written as $M_e$. $N$ is used to denote nonce. Finally, we write $\mathsf{Enc}_k(text)$ for the encryption of $text$, for which we assume indistinguishability under chosen plaintext attack. We also use $\mathsf{AEnc}_k(text)$ for authenticated encryption, for which we assume indistinguishability under chosen plaintext attack and integrity of ciphertext. We use cryptopp [1] to implement the above cryptographic operations, and we do not verify its implementation.

**One-time Password Generator** The abstract model of the OTP secret provisiong protocol (presented in detail in Section 2.3.1), where *client* runs in a SGX enclave, *bank* is a trusted remote service, and *disk* denotes client-side storage that is under adversary's control:

$$
\begin{aligned}
&bank \to client : N \\
&client \to bank : N \mid g^c \mid \mathsf{Quote}_{client}(N \mid g^c) \\
&bank \to client : N \mid g^b \mid \mathsf{Sig}_{K^{-1}_{bank}}(N \mid g^b) \mid \mathsf{AEnc}_{\mathsf{H}(g^{bc})}(secret) \\
&client \to disk : \mathsf{AEnc}_{K_{seal}}(secret)
\end{aligned}
$$

First, we use Moat to prove that $g^{bc}$ and $K_{seal}$ are not leaked to the attacker $M$. Next, ProVerif uses secrecy assumption on $g^{bc}$ and $K_{seal}$ to prove that *secret* is not leaked to a network adversary. This proof allows Moat to declassify *client*'s output to disk during type checking (Figure 7.5). Moat successfully proves that the *client* enclave satisfies confidentiality.

**Notary Service** We implement a notary service described in [56] but adapted to run on SGX. The notary enclave assigns logical timestamps to documents, giving them a total ordering. The notary enclave responds to (1) a connect message for obtaining the attestation report, and (2) a notarize message for obtaining a signature over the document hash and the current counter.

$$
\begin{aligned}
&user \to notary : \mathsf{connect} \mid N \\
&notary \to user : \mathsf{Quote}_{notary}(N) \\
&user \to notary : \mathsf{notarize} \mid \mathsf{H}(text) \\
&notary \to user : \mathsf{counter} \mid \mathsf{H}(text) \mid \\
&\qquad\qquad \mathsf{Sig}_{K^{-1}_{notary}}(\mathsf{counter} \mid \mathsf{H}(text))
\end{aligned}
$$

The only secret here is the private signature key $K^{-1}_{notary}$. First, we use Moat to prove that $K^{-1}_{notary}$ is not leaked to the attacker $M$. This proof fails because the output of Sig (in the response to notarize message) depends on the secret signature key — Moat is unaware of cryptographic properties of Sig. ProVerif proves that this message does not leak $K^{-1}_{notary}$ to a network adversary, which allows Moat to declassify this message and prove that the *notary* enclave satisfies confidentiality.

**End-to-End Encrypted Instant Messaging** We implement the off-the-record messaging protocol [27], which provides perfect forward secrecy and repudiability for messages exchanged between principals $A$ and $B$. We adapt this protocol to run on SGX, thus providing an additional guarantee that an infrastructure attack cannot compromise the ephemeral Diffie-Hellman keys, which encrypt and integrity-protect the messages between $A$ and $B$.

We only present a synchronous form of communication here for simplicity.

$$
\begin{aligned}
&A \to B : g^{a_1} \mid \mathsf{Sig}_{K_A^{-1}}(g^{a_1}) \mid \mathsf{Quote}_A(\mathsf{Sig}_{K_A^{-1}}(g^{a_1}))\\
&B \to A : g^{b_1} \mid \mathsf{Sig}_{K_B^{-1}}(g^{b_1}) \mid \mathsf{Quote}_B(\mathsf{Sig}_{K_B^{-1}}(g^{b_1}))\\
&A \to B : g^{a_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_1 b_1})}(m_1) \mid \mathsf{MAC}_{\mathsf{H}(\mathsf{H}(g^{a_1 b_1}))}(g^{a_2} \mid\\
&\qquad \mathsf{Enc}_{\mathsf{H}(g^{a_1 b_1})}(m_1))\\
&B \to A : g^{b_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2 b_1})}(m_2) \mid \mathsf{MAC}_{\mathsf{H}(\mathsf{H}(g^{a_2 b_1}))}(g^{b_2} \mid\\
&\qquad \mathsf{Enc}_{\mathsf{H}(g^{a_2 b_1})}(m_2))\\
&A \to B : g^{a_3} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2 b_2})}(m_3) \mid \mathsf{MAC}_{\mathsf{H}(\mathsf{H}(g^{a_2 b_2}))}(g^{a_3} \mid\\
&\qquad \mathsf{Enc}_{\mathsf{H}(g^{a_2 b_2})}(m_3))
\end{aligned}
$$

The OTR protocol only needs a digital signature on the initial Diffie-Hellman exchange — future exchanges use MACs to authenticate a new key using an older, known-authentic key. For the same reason, we only append a SGX quote to the initial key exchange. First, we use Moat to prove that the Diffie-Hellman secrets computed by $p_{enc}$ (i.e., $g^{a_1 b_1}$, $g^{a_2 b_1}$, $g^{a_2 b_2}$) are not leaked to the attacker $M$. Next, ProVerif uses this secrey assumption to prove that messages $m_1$, $m_2$, and $m_3$ are not leaked to the network adversary. The ProVerif proofs allows Moat to declassify all messages following the initial key exchange, and successfully prove confidentiality.

**Query Processing over Encrypted Table** In this case study, we evaluate Moat on a stand-alone application, removing the possibility of protocol attacks and therefore the need for any protocol verification. We build a database table containing two columns: name which is deterministically encrypted, and amount which is nondeterministically encrypted. A user wishes to select all rows with name "Alice" and sum all the amounts. We partition this computation into two parts: unprivileged computation (which selects the rows) and enclave computation (which computes the sum).

| Benchmark | x86+TAP instructions | BoogiePL statements | Moat proof | Policy Annotations |
|---|---|---|---|---|
| OTP | 188 | 1774 | 9.9 sec | 4 |
| Notary | 147 | 1222 | 3.2 sec | 2 |
| OTR IM | 251 | 2191 | 7.8 sec | 7 |
| Query | 575 | 4727 | 55 sec | 9 |

*Figure 7.7*: Summary of experimental results. Columns are (1) instructions analyzed by Moat (which does not include the cryptographic library), (2) size of $I(p_M)$, (3) proof time, (4) number of secret and declassifed annotations

## 7.4 Related Work

The work in this chapter relates three somewhat distinct areas in security.

**Secure Systems on Trusted Hardware.** In recent years, there has been growing interest in building secure systems on top of trusted hardware. Sancus [90] is a security architecture for networked embedded devices that seeks to provide security guarantees without trusting any infrastructural software, only relying on trusted hardware. Intel SGX [59] seeks to provide similar guarantees via extension to the x86 instruction set. There are some recent efforts on using SGX for trusted computation. Haven [21] is a system that exploits Intel SGX for shielded execution of unmodified Windows applications. It links the application together with a runtime library OS that implements the Windows 8 API. However, it does not provide any confidentiality or integrity guarantees, and includes a significant TCB. VC3 [107] uses SGX to run map-reduce computations while protecting data and code from an active adversary. However, VC3's confidentiality guarantee is based on the assumption that enclave code does not leak secrets, and we can use Moat to verify this assumption. Santos et al. [104] seek to build a trusted language runtime for mobile applications based on ARM TrustZone [9]. These design efforts have thrown up very interesting associated verification questions, and Moat seeks to address these with a special focus on Intel SGX and Sanctum.

**Verifying Information Flow on Programs.** Checking implementation code for safety is also a well studied problem. Type systems proposed by Sabelfeld et al. [101], Barthe et al. [19], and Volpano et al. [121] enable the programmer to annotate variables that hold secret values, and ensure that these values do not leak. Balliu et al. [13] automate information flow analysis of ARMv7 machine code, and Constanzo et al. [39] discuss verification of information flow security for C and Assembly programs. Languages and verification techniques also exist for quantitative information flow (e.g., [57]). However, these works assume that the infrastructure (OS/VMM, etc.) on which the code runs is safe, which is unrealistic due to malware and other attacks. Our approach builds upon this body of work, showing how it can be adapted to the setting where programs run on an adversarial OS/VMM, and instead rely on trusted hardware for information-flow security.

**Cryptographic Protocol Verification.** There is a vast literature on cryptographic protocol verification (e.g. [25, 26]). Our work builds on top of cryptographic protocol verifiers showing how to use them to reason about protocol attacks and to generate annotations for more precise verification of enclave programs. In the future, it may also be possible to connect our work to the work on correct-by-construction generation of cryptographic protocol implementation [51].

## 7.5 Summary

Despite the isolation guarantees that TAP offers against a privileged adversary, vulnerabilities in the enclave itself (e.g. incorrect use of TAP's primitives, memory safety errors, etc.) can be exploited to extract secrets from the enclave. In this chapter, we introduce the

first technique to formally verify properties of enclave programs. Specifically, we present a sound verification methodology (based on automated theorem proving and information flow analysis) for proving that an enclave program running on TAP does not contain a vulnerability that causes it to reveal secrets to the adversary. We introduce Moat, a tool which formally verifies confidentiality properties of applications running on the TAP. We evaluate Moat on several applications, including a one time password scheme, off-the-record messaging, notary service, and secure query processing.

# Chapter 8

# /CONFIDENTIAL : Scalably Verifying Confidentiality of Enclave's Outputs

In the previous chapter, we presented a verification technique, called Moat [112], to automatically verify that enclave programs satisfy confidentiality guarantees, even in the presence of a privileged adversary. While Moat implements a sound verification technique, it does not scale beyond enclaves containing few hundred x86 instructions — the inefficiency of Moat stems from global analysis of the entire enclave's code (i.e., lack of modular reasoning), and fine-grained tracking of the flow of secrets in enclave's memory at the level of machine code. Furthermore, Moat incurs a heavy manual effort as the user must annotate secrets within the machine code, and also declassify outputs when necessary. This chapter address the aforementioned scalability and automation concerns in Moat. Apart from Moat, other approaches to certifying confidentiality use programming languages that can express information-flow policies [44, 121, 100]. However, these approaches require use of particular programming languages, incur a heavy annotation burden, and place the compiler and the language runtime in the Trusted Computing Base (TCB) [103]. This chapter also explores certifying the machine code loaded into enclaves (similar to Moat) to avoid these trust dependencies. In short, the goal of this chapter to guarantee confidentiality properties of enclave programs, but with far greater scalability and automation compared to previous approaches, and a tiny TCB (containing First-Order Logic decision procedures).

We propose to verify a specific confidentiality policy: the code inside the enclave can perform arbitrary computations within the region, but it can only generate output data through an encrypted channel. We refer to this property as *Information Release Confinement* or *IRC*. This is a meaningful confidentiality policy because it guarantees that attackers can only observe encrypted data. We exclude covert channels and side channels (e.g., timing, power) from our threat model. Previous experience from building sensitive data analytics services using enclaves suggests that IRC is not unduly restrictive. For example, in VC3 [107], only map and reduce functions are hosted in enclaves; the rest of the Hadoop stack is untrusted. Both mappers and reducers follow a stylized idiom where they receive encrypted input from Hadoop's untrusted communication layer, decrypt the input, process it, and send

encrypted output back to Hadoop. No other interaction between these components and the untrusted Hadoop stack is required.

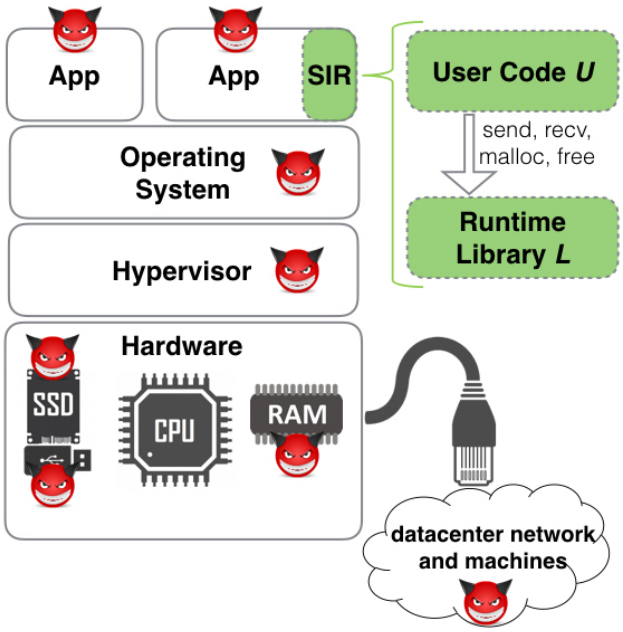



*Figure 8.1*: Recall the threat model: The adversary $M$ controls the host OS, hypervisor, and any hardware beyond the CPU package, which may include storage devices. The adversary also controls all other machines and the network. The enclave is the only trusted software component.

Scalably verifying IRC is challenging, because we aim to have a verification procedure that can automatically certify machine code programs, without assuming the code to be type safe or memory safe; for example, programs may have exploitable bugs or they may unintentionally write information out of the enclave through corrupted pointers. Our approach is based on decomposing programs into a user application ($U$) that contains the application logic and a small runtime library ($L$) that provides core primitives such as memory management and encrypted channels for communication. We restrict the interaction between the user application and the untrusted platform to the narrow interface implemented by $L$.

The main contributions of this chapter are: (1) a design methodology to program enclaves using a narrow interface to a library, (2) a notion called Information Release Confinement (IRC), which allows separation of concerns while proving confidentiality in the presence of a privileged adversary (that controls the OS, hypervisor, etc.), and (3) a modular and scalable verification method for automatically checking IRC directly on application binaries.

This chapter has the following organization. In Section 8.1, we give present the overall approach and key ideas of /Confidential . Section 8.2 discusses the relationship between IRC and confidentialty (which we defined in Chapter 6), and shows how a proof of IRC can be decomposed into properties of $U$ and $L$. We discuss the methodology for static verification of $U$ code in Section 8.3, and several aspects of the implementation of /Confidential

in Section 8.4. Next, we evaluate /Confidential on several large enclave programs in Section 8.5. Finally, we discuss several related works in Section 8.6.

## 8.1 Overview of /Confidential

/Confidential decomposes the enclave program into a user application ($U$), written by the developer, and runtime library ($L$) that is written once and for all. /Confidential mandates that $U$ satisfy IRC, which restricts the interaction between $U$ and the untrusted platform to the narrow interface implemented by $L$ — $L$ implements primitives for secure communication with remote parties, cryptographically protected file storage, memory management, etc. A key contribution of /Confidential is how this methodology enables decomposing the confidentiality verification in two parts. For $L$, we need to verify that it implements a secure encrypted channel and memory management correctly; $L$ is a small library that can be written and verified once and for all. For $U$, we need to verify a series of constraints on memory loads, stores, and control-flow transitions. Specifically, we need to check that stores do not write data to non-enclave memory (since all interaction with the untrusted platform must go through $L$), stores do not corrupt control information (e.g., return addresses and jump tables) inside the enclave, stores do not corrupt the state of $L$, loads do not read cryptographic state from $L$ (since using cryptographic state in $U$ could break the safety of the encrypted channel [24]), calls go to the start of functions in $U$ or to the entry points of API functions exported by $L$, and jumps target legal (i.e., not in the middle of) instructions in the code. These checks on $U$ are a weak form of control-flow integrity, and along with restrictions on reads and writes, enforce a property which we call WCFI-RW. We show that the functional correctness of $L$ combined with WCFI-RW of $U$ implies our desired confidentiality policy (IRC).

This chapter formalizes WCFI-RW and propose a two-step process to automatically verify that an application satisfies WCFI-RW. We first use a compiler that instruments $U$ with runtime checks [107]. Next, we automatically verify that the instrumentation in the compiled binary is sufficient for guaranteeing WCFI-RW. Our verifier generates verification conditions from the machine code of the application, and automatically discharges them using an SMT solver. This step effectively removes the compiler as well as third-party libraries from the TCB. By verifying these libraries, users can be certain that they do not leak information either accidentally, through exploitable bugs, or by intentionally writing data out of the enclaves.

This approach is significantly different from verifying arbitrary user code with unconstrained interfaces for communication with the untrusted platform. We require no annotations from the programmer — all of $U$'s memory is considered secret. The TCB is small — it includes the verifier but does not include $U$ or the compiler. Furthermore, the assertions required to prove WCFI-RW are simple enough to be discharged using an off-the-shelf Satisfiability Modulo Theories (SMT) solver. The verification procedure is modular, and can be done one procedure at a time, which enables the technique to scale to large programs. Our

experiments suggest that our verifier can scale to real-world applications, including map-reduce tasks from VC3 [107]. In the remainder of this section, we describe the challenges addressed and solutions adopted by /CONFIDENTIAL.

```
1 void Reduce(BYTE *keyEnc, BYTE *valuesEnc,
2             BYTE *outputEnc) {
3   KeyAesGcm *aesKey = ProvisionKey();
4
5   char key[KEY_SIZE];
6   aesKey->Decrypt(keyEnc, key, KEY_SIZE);
7
8   char valuesBuf[VALUES_SIZE];
9   aesKey->Decrypt(valuesEnc, valuesBuf, VALUES_SIZE);
10  StringList *values = (StringList *) valuesBuf;
11
12  long long usage = 0;
13  for (char *value = values->begin();
14            value != values->end();
15            value = values->next()) {
16    long lvalue = mystrtol(value, NULL, 10);
17    usage += lvalue;
18  }
19
20  char cleartext[BUF_SIZE];
21  sprintf(cleartext, "%s %lld", key, usage);
22  aesKey->Encrypt(cleartext, outputEnc, BUF_SIZE);
23 }
```

(a) Sample reducer method

```
1 void Reduce(Channel<char*>& channel) {
2   char key[KEY_SIZE];
3   channel.recv(key, KEY_SIZE);
4
5   char valuesBuf[VALUES_SIZE];
6   channel.recv(valuesBuf, VALUES_SIZE);
7   StringList *values = (StringList *) valuesBuf;
8
9   long long usage = 0;
10  for (char *value = values->begin();
11            value != values->end();
12            value = values->next()) {
13    long lvalue = mystrtol(value, NULL, 10);
14    usage += lvalue;
15  }
16
17  char cleartext[BUF_SIZE];
18  sprintf(cleartext, "%s %lld", key, usage);
19  channel.send(cleartext);
20 }
```

(b) Reducer method using secure channels

*Figure 8.2*: Reducer method illustrating the challenges of proving confidentiality.

### 8.1.1 Verifying confidentiality.

We illustrate the challenges in verifying that code running inside an enclave satisfies confidentiality. Consider the `Reduce` method in Figure 8.2, which acts as a reducer in a map-reduce job and is implemented within an enclave. To protect sensitive data during the computation, the data remains encrypted outside an enclave (i.e., the map-reduce framework, such as Hadoop, never observes sensitive data in cleartext) and is only decrypted within the mapper and reducer functions which are implemented within enclaves. The reducer receives encrypted a key and list of values from different mappers. The method first provisions an encryption key (of type `KeyAesGcm`, not to be confused with the data key) by setting up a secure channel with a component that manages keys (not shown for brevity). It decrypts the key and values, computes the sum of all values, and writes the output to a buffer. The buffer is encrypted and written to a location outside the enclave specified by the map-reduce framework.

Proving that this `Reduce` method preserves confidentiality is challenging. The code writes the result of the computation to a stack-allocated buffer without checking the size of the inputs. This may result in a vulnerability that can be exploited to overwrite the return address, execute arbitrary code and leak secrets. Therefore, the proof must show that the

cumulative size of key and result does not exceed the buffer size. Such a proof may require non-modular reasoning since the sizes may not be defined locally. Furthermore, `Reduce` writes to a location outside the enclave. The proof must show that the data written is either encrypted or independent of secrets. The latter requires precise, fine-grained tracking of secrets in the enclave's memory. Also, unrelated to the application logic, we note that the `Reduce` method manually provisions its encryption keys. Therefore, a proof of confidentiality must also show that the key exchange protocol is secure, and that the keys are neither leaked by the application, nor overwritten by an adversary. Finally, the proof must also show that the compilation to machine code preserves semantics of source code. Therefore, building a scalable and automatic verifier for confidentiality is challenging for arbitrary user code.

### 8.1.2 Restricted interface.

We propose a design methodology to separate various concerns in ensuring confidentiality, and enable simpler and scalable tools to automatically verify confidentiality. In our methodology, the user application $U$ is statically linked with a runtime library $L$ that supports a narrow communication interface. During initialization, the runtime is configured to setup a secure channel with another trusted entity and provision secrets, e.g., the encryption key. At runtime, $U$ can use $L$'s APIs to allocate memory (via `malloc` and `free`) and send or receive data over the secure channel (via `send` and `recv`). Figure 8.2(b) shows the `Reduce` method that has been rewritten to use this interface. The method calls `recv` to retrieve data from the channel, which reads and decrypts encrypted values from outside the enclave. After computing the result, the method calls `send` to write data to the channel, which internally encrypts the data and writes it to a location outside the enclave. Observe that there are no writes to non-enclave memory directly from this `Reduce` method.

Restricting the application to this interface serves an important purpose — it allows us to decompose the task of verifying confidentiality into two sub-tasks: (1) checking that the user application $U$ communicates with the external world *only* through this interface, and (2) checking that the implementation of the interface in $L$ does not leak secrets. /CONFIDENTIAL implements verification of Information Release Confinement (IRC): $U$ can only write to non-SIR memory by invoking the `send` API. Proving information leakage properties of the implementation of `send` (e.g., strong encryption property, resistance to various side channels, etc.) would require one-time human-assisted verification of the library code $L$, and is left for future work. We feel that IRC is an important stepping stone for achieving confidentiality guarantees of enclaves against various adversaries.

We do not find this narrow interface to be restrictive for our target applications: trusted cloud services, which are typically implemented as a collection of distributed and trusted entities. In this setting, the application ($U$) only sends and receives encrypted messages from remote parties. We use this approach in practice to build sensitive data analytics, database services, key manager, etc.

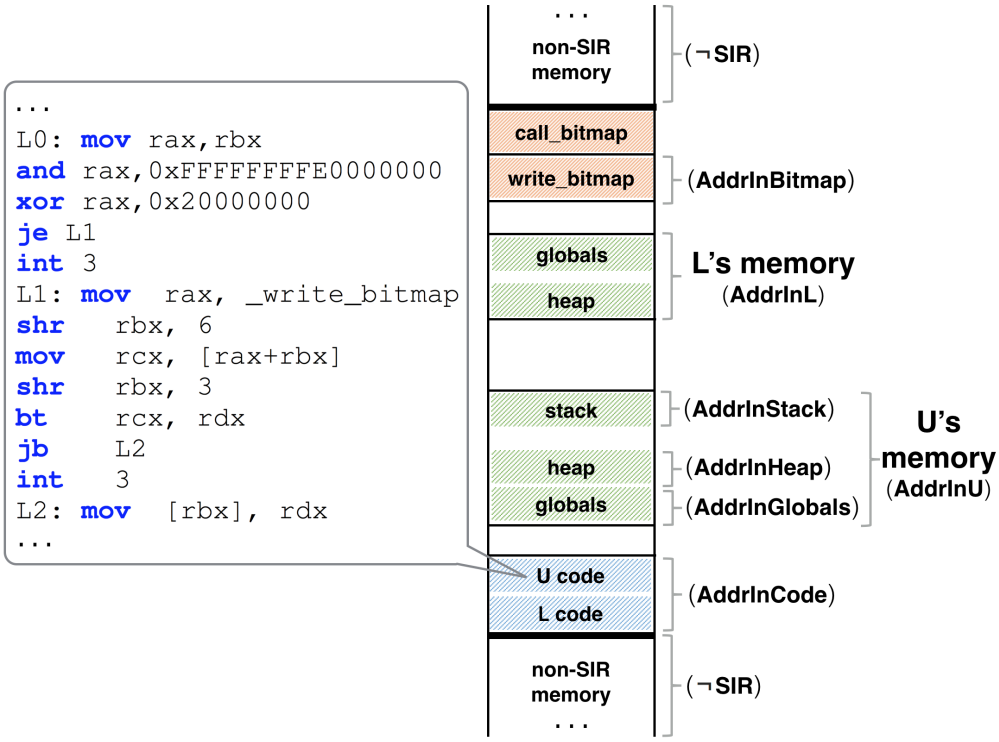



*Figure 8.3*: Memory layout of enclave and instrumentation sequence for unsafe stores in the application code ($U$).

### 8.1.3 Checking IRC.

For scalable and automatic verification, we further decompose IRC into a set of checks on $U$ and contracts on each of the APIs in $L$. First, we verify that $U$ satisfies WCFI-RW: $U$ transfers control to $L$ only through the API entry points, and it does not read $L$'s memory or write outside $U$'s memory. Apart from the constraints on reads and writes, WCFI-RW requires a weak form of control-flow integrity in $U$. This is needed because if an attacker hijacks the control flow of $U$ (e.g., by corrupting a function pointer or return address), then $U$ can execute arbitrary instructions, and we cannot give any guarantees statically. Next, we identify a small contract on $L$ that, in conjunction with WCFI-RW property of $U$, is sufficient for proving IRC. The contracts on $L$ (defined in section 8.2.2) ensure that $L$ does not modify $U$'s state to an extent that WCFI-RW is compromised (e.g. by tampering code pointers within $U$'s stack frame).

From here on, we describe how we check WCFI-RW directly on the machine code of $U$, which enables us to keep the compiler and any third-party application libraries out of the TCB. Third-party libraries need to be compiled with our compiler, but they can be shipped in binary form. Furthermore, to help the verification of WCFI-RW, our (untrusted) compiler adds runtime checks to $U$'s code, and we use the processor's paging hardware to protect $L$'s memory — although the attacker controls the page tables, the processor protects page table

entries that map to enclave's memory. We use the same runtime checks on memory writes and indirect calls as VC3 [107], but we implement efficient checks on memory reads using the paging hardware.

Next, we describe the checks instrumented in the binary by the compiler. We first note that $U$ and $L$ share the same address space (Figure 8.3). The code segments of both $U$ and $L$ are placed in executable, non-writable pages. The region also contains a stack shared by both $U$ and $L$. We isolate $L$ from $U$ by using a separate heap for storing $L$'s internal state. The compiler enforces WCFI-RW by instrumenting $U$ with the following checks:

- **Protecting return addresses**: To enforce that writes through pointers do not corrupt return addresses on the stack, the compiler maintains a bitmap (`write_bitmap` in Figure 8.3) to record which areas in U's memory are writable. The `write_bitmap` is updated at runtime, while maintaining the invariant that a return address is never marked writable. Address-taken stack variables are marked writable by inlined code sequences in function prologues, and heap allocations and address-taken globals are marked writable by the runtime library. `store` instructions are instrumented with an instruction sequence (instructions from label L1 to L2 in Figure 8.3) that reads the `write_bitmap` and terminates the enclave program if the corresponding bit is not set. Note that the bitmap protects itself: the bits corresponding to the bitmap are never set.

- **Protecting indirect control flow transfers**: The compiler maintains a separate bitmap (`call_bitmap` in Figure 8.3) that records the entry points of procedures in $U$ and APIs of $L$. The compiler instruments indirect calls with an instruction sequence that reads the bit corresponding to the target address, and terminates the enclave program if that bit is unset. Indirect jumps within the procedure are also checked to prevent jumps to the middle of instructions; the checks consist of a range check on indices into jump tables (which are stored in read-only memory).

- **Preventing writes outside enclaves**: The compiler adds range checks to prevent writes to non-enclave memory (instructions from label L0 to L1 in Figure 8.3).

We note that the compiler's instrumentation offers a stronger guarantee than IRC, and prevents several forms of memory corruption inside the SIR. Specifically, the checks guarantee: 1) integrity of all data that is not address-taken, 2) detection of all sequential overflows and underflows on heap and stack, 3) integrity of return addresses, and 4) forward edge CFI. Future work may be able to verify of all these properties, but for this work we focus on verifying that the compiler's instrumentation is sufficient to guarantee WCFI-RW, because WCFI-RW together with the correctness of $L$ implies IRC (as we show in Theorem 1), and IRC is a strong property that provides meaningful security guarantees.

Preventing $U$ from accessing $L$'s memory is also important because $L$ keeps cryptographic secrets; allowing $U$ to read such secrets could break the typical assumptions of encryption algorithms (e.g., a key should not be encrypted with itself [24]). We achieve this efficiently

by requiring $L$ to store any such secrets in its own separate heap, and using TAP's memory protections to set page permissions to disable read/write access before transferring control to $U$. Note that we cannot use page permissions to prevent writes to non-enclave memory, because those page tables are controlled by a privileged adversary, e.g., kernel-level malware.

Let's reconsider the `Reduce` method in Figure 8.2(b). Compiling this method with the preceding runtime checks and necessary page permissions ensures that any vulnerability (e.g., line 17) cannot be used to violate WCFI-RW. Any such violation causes the program to halt.

To achieve good performance, the compiler omits runtime checks that it can statically prove to be redundant (e.g., writes to local stack-allocated variables). However, compilers are large code bases and have been shown to have bugs and produce wrong code at times [128, 83, 68, 77, 20]. The rest of this chapter describes our approach for automatically verifying that the machine code of $U$ satisfies WCFI-RW, thereby removing the compiler from the TCB. We also show that such a $U$ satisfies IRC when linked with an $L$ that satisfies our contract (defined in section 8.2.2).

## 8.2 Decomposing Proof of Confidentiality

In this section, we discuss how to verify confidentiality of an enclave program, as we defined in Definition 3 within Chapter 6. Since we must reason about $U$'s behaviors in the presence of a privileged adversary $M$ (defined in Section 4.2), /Confidential instruments $M$'s havocing operations to non-enclave memory, in the same method as Moat (see Section 6.1). We call this instrumented model $U_M$, and perform verification of $U_M$.

Confidentiality, as we defined in Definition 3, is expressed as a hyper-property [36], where we require that the adversary cannot infer secret state based on observations of non-enclave memory. Automatically checking if a program satisfies confidentiality usually requires precise tracking of the memory locations that may contain secrets during execution, which is the algorithm that we implemented within Moat. This is accomplished with either a whole-program analysis of $U_M$, which is hard to scale for machine code, or fine-grained annotations that specify locations with secrets, which is cumbersome for machine code [112]. We follow a different approach in order to scale the verification to large programs without requiring any annotation.

We expect $U_M$ to communicate with non-enclave entities, but follow a methodology where we mandate all communication to occur via the `send` and `recv` APIs in $L$. We require (and verify) that $U_M$ does not write to non-enclave memory. Instead, $U_M$ invokes `send`, which takes as an argument a pointer to the input buffer, and encrypts and integrity-protects the buffer before copying out to non-enclave memory. This methodology leads to a notion called *information release confinement* (IRC), which mandates that the only operations in $U_M$ that update non-enclave memory (apart from $M$'s havoc operations) are the `call send` statements. In other words, the only I/O interaction between the enclave and the attacker-controlled state occurs via the trustworthy runtime library $L$, which enforces cryptographic

protections on the I/O.

**Definition 5 Information Release Confinement.** *An execution trace* $[\sigma_0, \ldots, \sigma_n]$ *of* $U_M$ *satisfies information release confinement or IRC, if all updates to the adversary observable state (i.e.,* $\texttt{mem}_{\neg\texttt{evrange}}$*) in* $[\sigma_0, \ldots, \sigma_n]$ *are caused by either* `call send` *operations or* $\mathsf{havoc}_{\neg\texttt{evrange}}$ `mem` *operations (from* $M$*), i.e.,* $IRC([\sigma_0, \ldots, \sigma_n])$ *iff:*

$$\begin{aligned}
&\forall\, i \in \{0, \ldots, n-1\}\ .\\
&\quad (\texttt{stmt}(\sigma_i) \neq \texttt{call send}\ \wedge \texttt{stmt}(\sigma_i) \neq \mathsf{havoc}_{\neg\texttt{evrange}}\ \texttt{mem}) \Rightarrow\\
&\quad (\forall\, a.\ \neg\texttt{evrange}(a) \Rightarrow \sigma_i(\texttt{mem})[a] = \sigma_{i+1}(\texttt{mem})[a]) \qquad (8.1)
\end{aligned}$$

*$U_M$ satisfies IRC iff all traces of $U_M$ satisfy IRC.*

**IRC and Confidentiality.** IRC is an important building block in guaranteeing confidentiality, as defined in Definition 3. IRC ensures that, in any execution, the only outbound communication with the environment is via `send`. Hence, we can arrange for `send` to encrypt all its received data before transmitting it, to prevent explicit information leaks. In order for the encrypted data to be confidential, we additionally need to ensure that the encryption key in $L$ does not leak or gets overwritten. The definition of IRC enables us to separate properties we require from the application code $U_M$, and properties we require from $L$, in order to guarantee confidentiality.

It is important to note that IRC is not sufficient for protecting secrets from all side channels. Observations of the number and timing of `send` invocations, memory access patterns, electromagnetic radiation, etc. potentially reveal secrets. Nevertheless, if an application $U_M$ satisfies IRC, then we can eliminate certain channels and obtain various degrees of confidentiality by imposing additional constraints on the implementation of `send`. We can arrange for `send` to output messages with constant-sized buffers (using appropriate padding) to prevent the adversary from making any inference based on message sizes. In addition, we can arrange for `send` to do internal buffering and produce sequences of output messages that are separated by an interval that is independent of secrets, to prevent the adversary from making any inference based on timing. These defenses impose additional constraints only on the implementation of `send`. We plan to explore guaranteeing such properties of our `send` implementation in future work.

In the remainder of this section, we formalize the properties on both $U_M$ and $L$, such that the enclave satisfies the IRC property. To decouple the verification, we decompose IRC into 1) checking WCFI-RW of $U_M$, and 2) checking correctness properties of $L$'s API implementation.

### 8.2.1 WCFI-RW Property of $U_M$

WCFI-RW further decomposes into the following two properties:

(a) A weak form of control flow integrity (CFI) of $U_M$. A trace of $U_M$ satisfies weak CFI if 1) each `call` statement in the trace targets the starting address of a procedure in

$U$ or API entry point of $L$, 2) each ret statement in the trace uses the return address saved by the matching `call` statement in that trace, 3) each `jmp` statement in the trace targets a legal instruction within the procedure or the starting address of a procedure in $U$'s code.

(b) $U_M$ does not read from or write to $L$'s memory, and does not write to non-enclave memory.

WCFI-RW guarantees that $U_M$ only calls into $L$ at allowed API entrypoints, which allows us to soundly decouple the verification of $U_M$ from $L$. WCFI-RW prevents a large class of CFI attacks (e.g., ROP attacks): backward control edges (returns) are fully protected, and forward edges (calls and jumps) are significantly constrained. Furthermore, observe that by preventing jumps into the middle of instructions, we guarantee that the code of $U_M$ that we statically verify is same that will execute at runtime. However, this form of CFI is weaker than standard CFI [4] because it allows a procedure in $U_M$ to call any other procedure in $U_M$. In other words, a program that satisfies WCFI-RW may exhibit control transfers that are not present in the source program, and this can bootstrap certain control-flow attacks. Nevertheless, for any such attack that is not blocked by WCFI-RW, we prove that they still cannot break IRC (soundness theorem in section 8.2.3); in the end, the attacker only observes encrypted values.

To formalize WCFI-RW, we construct a monitor automaton $\mathcal{W}$ (defined next) from $U_M$ to check whether $U_M$ satisfies WCFI-RW, similar in spirit to prior works on CFI [4, 50]. $\mathcal{W}$ is synchronously composed with $U_M$, such that the statements executed by $U_M$ form the input sequence of $\mathcal{W}$. We say that WCFI-RW is violated whenever $\mathcal{W}$ reaches a stuck state during the execution of $U_M$. The formalization of WCFI-RW requires the following predicates over addresses in the region (illustrated in Figure 8.3). For any enclave address $a$, $\mathsf{AddrInHeap}(a)$ is true if $a$ belongs to $U$'s heap. $\mathsf{AddrInStack}(a)$ is true if $a$ belongs to the enclave's stack (which is shared by both $U$ and $L$). $\mathsf{AddrInU}(a)$ is true if $a$ belongs to $U$'s memory (stack, globals, or heap), and $\mathsf{AddrInL}(a)$ is true if $a$ belongs to $L$'s memory (globals or heap). $\mathsf{AddrInCode}(a)$ is true if $a$ belongs to the enclave's code (either $U$ or $L$'s code). Finally, $\mathsf{writable}(\texttt{mem}, a)$ is true iff the bit corresponding to address $a$ is set in the `write_bitmap`.

**Definition 6 WCFI-RW Monitor Automaton**
*$\mathcal{W} = (\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a generalized pushdown automaton where $\mathcal{Q} = \{\sigma\}$ is its set of states, $\Sigma = \{$ `call` $e$, ret, `jmp` $e$, v $:=$ $e$, v $:=$ $\mathsf{load_n}(e_a)$, $\mathsf{store_n}(e_a, e_d)$, $\mathsf{havoc}_{\neg\mathsf{evrange}}$ `mem` $\}$ is its set of inputs, $\Gamma = \{a \mid \mathsf{AddrInCode}(a)\}$ is its finite set of stack symbols, $q_0 = \sigma_{entry}$ is its initial state ($\sigma$entry being the machine state following the jump from $L$ into $U$'s entry point), $Z_0 = a_{entry}$ is its initial stack, $F = Q$ is its set of accepting states, $\delta$: $(\mathcal{Q} \times \Sigma \times \Gamma^*) \to \mathcal{P}(\mathcal{Q} \times \Gamma^*)$ is its transition function. Furthermore, let $\mathsf{next}(\mathsf{pc})$ be the address of the subsequent instruction in $U$ after decoding the instruction starting at address $\mathsf{pc}$, and $\sigma'$ be the state resulting from executing an instruction $i \in Instr$ starting in $\sigma$, i.e.,*

$(\sigma, \sigma') \in \rightsquigarrow$ *(as per the operational semantics in Figure 3.3). The transition function* $\delta$ *of the monitor automaton* $\mathcal{W}$ *is as follows:*

$$\delta(\sigma, \texttt{call}\ e, \gamma) = \begin{cases} \{(\sigma', \mathsf{next}(\sigma(\mathsf{pc})) \cdot \gamma)\} & \textit{iff } \psi_{\texttt{call}} \\ \emptyset & \textit{otherwise} \end{cases}$$

*where* $\psi_{\texttt{call}} \doteq \sigma(e)$ *is the address of a procedure entry in* $U$

$$\delta(\sigma, \texttt{call}\ \mathsf{x}, \gamma) = \{(\sigma', \gamma)\}$$

*where* $\mathsf{x} \in \{\mathsf{malloc}, \mathsf{free}, \mathsf{send}, \mathsf{recv}\}$

$$\delta(\sigma, \mathsf{ret}, a \cdot \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff } \sigma(\mathsf{mem})[\sigma(\mathsf{rsp})] = a \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\delta(\sigma, \texttt{jmp}\ e, \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff } \psi_{\texttt{jmp}} \\ \emptyset & \textit{otherwise} \end{cases}$$

*where* $\psi_{\texttt{jmp}} \doteq \psi_{\texttt{call}} \vee$ $\sigma(e)$ *is an instruction in current procedure*

$$\delta(\sigma, \mathsf{flag} := e, \gamma) = \{(\sigma', \gamma)\} \textit{ where } \mathsf{flag} \in \mathsf{flags}$$

$$\delta(\sigma, \mathsf{rsp} := e, \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff } \psi_{\mathsf{rsp}} \\ \emptyset & \textit{otherwise} \end{cases}$$

*where* $\psi_{\mathsf{rsp}} \doteq \mathsf{AddrInU}(\sigma(e))$

$$\delta(\sigma, \mathsf{reg} := e, \gamma) = \{(\sigma', \gamma)\} \textit{ where } \mathsf{reg} \in \mathsf{regs} \setminus \{\mathsf{rsp}\}$$

$$\delta(\sigma, \mathsf{reg} := \mathsf{load}_{\mathsf{n}}(e_a), \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff } \psi_{\mathsf{load}} \\ \emptyset & \textit{otherwise} \end{cases}$$

*where* $\psi_{\mathsf{load}} \doteq \neg\mathsf{AddrInL}(\sigma(e_a)) \wedge \neg\mathsf{AddrInL}(\sigma(e_a) + n - 1)$

$$\delta(\sigma, \mathsf{store}_{\mathsf{n}}(e_a, e_d), \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff } \psi_{\mathsf{store}} \\ \emptyset & \textit{otherwise} \end{cases}$$

*where* $\psi_{\mathsf{store}} \doteq \mathsf{AddrInU}(\sigma(e_a)) \wedge \mathsf{AddrInU}(\sigma(e_a) + n - 1)$

$$\delta(\sigma, \mathsf{havoc}_{\neg\mathsf{evrange}}\ \mathtt{mem}, \gamma) = \{(\sigma', \gamma)\}$$

**Definition 7 WCFI-RW**
*WCFI-RW is violated in an execution trace* $[\sigma_0, \ldots, \sigma_n]$ *when no transition exists in* $\mathcal{W}$ *for an instruction in* $[\sigma_0, \ldots, \sigma_n]$*, i.e.,* $\mathcal{W}$ *gets stuck. Formally, WCFI-RW*$([\sigma_0, \ldots, \sigma_n])$ *iff (with starting state* $\sigma 0 = \sigma_{entry}$ *and initial stack* $\gamma_0 = a_{entry}$*):*

$$\exists \gamma_0, \ldots, \gamma_n \in \Gamma^* \,.\, \bigwedge_{k=0}^{n-1} (\sigma_{k+1}, \gamma_{k+1}) \in \delta(\sigma_k, \texttt{instr}(\sigma_k), \gamma_k)$$

$U_M$ *satisfies WCFI-RW if all traces of* $U_M$ *satisfy WCFI-RW.*

The role of the pushdown stack in Definition 6 is to match the calls and returns. $\mathcal{W}$ only modifies the pushdown stack on `call` and ret statements, and updates the state as per the operational semantics defined in Figure 3.3. We now describe each case in the definition of the transition function $\delta$ of $\mathcal{W}$. On a `call` to a procedure in $U$, $\mathcal{W}$ pushes the return address (i.e., the address of the subsequent instruction) onto the pushdown stack, for use by the ret statement. On a `call` to $L$'s API, since $L$ only contributes one step to the trace, and since correctness of $L$'s APIs (section 8.2.2) guarantees that the `call` returns to the call site, $\mathcal{W}$ does not push the return address onto the pushdown stack. A ret produces a valid transition only when the topmost symbol on the pushdown stack matches the return address on the machine's stack; this transition pops the topmost element off the pushdown stack. A `jmp` produces a valid transition if it targets a legal instruction in the current procedure, or the beginning of a procedure in $U$. Assignment to rsp succeeds if the new value is an address in $U$'s memory — this constraint is needed because `call` and ret accesses `mem` at address rsp, and WCFI-RW requires stores to be contained within $U$'s memory. Other registers and flags can be assigned to arbitrary values. Finally, to satisfy WCFI-RW's constraints on reads and writes, a `load` succeeds iff the address is not within $L$'s memory, and a `store` succeeds iff the address is within $U$'s memory.

### 8.2.2 Correctness of L's API Implementation

While we strive for full functional correctness of $L$, the following contract (in conjunction with WCFI-RW of $U_M$) is sufficient for proving IRC of the enclave.

(a) `malloc(size)` (where the return value `ptr` is the starting address of the allocated region) must not 1) modify non-enclave memory or stack frames belonging to $U$, 2) make any stack location writable, or 3) return a region outside $U$'s heap. Formally, when $\texttt{instr}(\sigma) = \texttt{call}$ malloc, we write $\sigma \rightsquigarrow \sigma'$ with the following conditions:

- ▷ $\forall a.\ (\neg \mathsf{evrange}(a) \ \vee \ (\mathsf{AddrInStack}(a) \ \wedge \ a \geq \sigma(\mathsf{rsp}))) \Rightarrow \sigma(\mathsf{mem})[a] = \sigma'(\mathsf{mem})[a]$
- ▷ $\forall a.\ \mathsf{AddrInStack}(a) \Rightarrow (\mathsf{writable}(\sigma(\mathsf{mem}),\ a) \Leftrightarrow \mathsf{writable}(\sigma'(\mathsf{mem}),\ a))$
- ▷ $\sigma'(\mathsf{ptr}) = 0 \ \vee \ ((\sigma'(\mathsf{ptr}) \leq \sigma'(\mathsf{ptr}) + \sigma(\mathsf{size})) \ \wedge$
  $\mathsf{AddrInHeap}(\sigma'(\mathsf{ptr})) \ \wedge \mathsf{AddrInHeap}(\sigma'(\mathsf{ptr}) + \sigma(\mathsf{size})))$

First, by forbidding `malloc` from modifying $U$'s stack above rsp, we prevent `malloc` from overwriting return addresses in $U$'s stack frames. Second, by forbidding `malloc` from making a stack location writable, we prevent a return address from being corrupted later by code in $U$ — `malloc` should only modify the `write_bitmap` to make the allocated region writable. Both restrictions are paramount for preventing WCFI-RW exploits. Finally, we require `malloc` to return a region from $U$'s heap (or the null pointer), and ensure that a machine integer overflow is not exploited to violate IRC.

(b) `free(ptr)` must not 1) modify non-enclave memory or stack frames belonging to $U$, or 2) make any stack location writable. Formally, when $\texttt{instr}(\sigma) = \texttt{call free}$, we write $\sigma \rightsquigarrow \sigma'$ with the following conditions:

- ▷ $\forall a.\ (\neg\mathsf{evrange}(a) \ \vee\ (\mathsf{AddrInStack}(a) \ \wedge\ a \geq \sigma(\mathsf{rsp}))) \Rightarrow \sigma(\mathsf{mem})[a] = \sigma'(\mathsf{mem})[a]$
- ▷ $\forall a.\ \mathsf{AddrInStack}(a) \Rightarrow (\mathsf{writable}(\sigma(\mathsf{mem}),\ a) \Leftrightarrow \mathsf{writable}(\sigma'(\mathsf{mem}),\ a))$

These constraints are equivalent to the constraints on `malloc`, and are likewise paramount for preventing WCFI-RW exploits. Note that we do not require `malloc` to return a previously unallocated region, nor do we require `free` to mark the freed region as invalid; full functional correctness would require such properties. WCFI-RW does not assume any invariants on the heap values, and therefore vulnerabilities such as use-after-free do not compromise WCFI-RW.

(c) `send(ptr,size)` must not make any address writable or modify the stack frames belonging to $U$. Formally, when $\texttt{instr}(\sigma) = \texttt{call send}$, we write $\sigma \rightsquigarrow \sigma'$ with the following conditions:

- ▷ $\forall a.\ (\mathsf{AddrInBitmap}(a) \vee (\mathsf{AddrInStack}(a) \wedge a \geq \sigma(\mathsf{rsp}))) \Rightarrow \sigma(\mathsf{mem})[a] = \sigma'(\mathsf{mem})[a]$

`send` is used to encrypt and sign the message buffer before writing to non-enclave memory, and it is the only API call that is allowed to modify non-enclave memory. However, we forbid `send` from modifying a caller's stack frame or the bitmap. By preventing `send` from modifying $U$'s stack above rsp, we prevent `send` from overwriting a return address in any of $U$'s stack frames. Furthermore, `send` cannot modify the bitmap and make any location writable, thereby preventing a return address from being modified later by some code in $U$.

(d) `recv(ptr,size)` must 1) check that the destination buffer is a writable region in $U$'s memory, and 2) not modify any memory location outside the input buffer. Formally, when $\texttt{instr}(\sigma) = \texttt{call recv}$, we write $\sigma \rightsquigarrow \sigma'$ with the following conditions:

- ▷ $\forall a.\ (\sigma(\mathsf{ptr}) \leq a < \sigma(\mathsf{ptr}) + \sigma(\mathsf{size})) \Rightarrow (\mathsf{writable}(\sigma(\mathsf{mem}),a)\ \wedge \mathsf{AddrInU}(a))$
- ▷ $\forall a.\neg(\sigma(\mathsf{ptr}) \leq a < \sigma(\mathsf{ptr}) + \sigma(\mathsf{size})) \Rightarrow \sigma(\mathsf{mem})[a] = \sigma'(\mathsf{mem})[a]$
- ▷ $\sigma(\mathsf{ptr}) \leq \sigma(\mathsf{ptr}) + \sigma(\mathsf{size})$

`recv` is used to copy an encrypted, signed message from non-enclave memory, decrypt it, verify its signature, and copy the cleartext message buffer to $U$'s memory. The first two constraints ensure that the message is written to a writable memory region within $U$ (which guarantees that a return address is not modified by `recv`), and that the cleartext message is not written out to non-enclave memory. The final constraint ensures that an integer overflow is not exploited to violate IRC.

In addition to the contracts above, we check the following contracts for each of `malloc`, `free`, `send`, and `recv`:

- page permissions, following the API call, are set to prevent read and write access to $L$'s memory. Write access is disabled to prevent $U$ from corrupting $L$'s state, whereas read access is disabled to prevent reading $L$'s secrets.
- stack pointer `rsp` is restored to its original value.
- the API call satisfies the application binary interface (ABI) calling convention (Windows x64 in our implementation)

We assume that the implementation of $L$ satisfies the above contracts. Since $L$ is written once, and used inside all enclaves, we could potentially verify the implementation of $L$ once and for all manually.

### 8.2.3 Soundness

**Theorem 3** *If $U_M$ satisfies WCFI-RW and the implementation of L's API satisfies the correctness properties given in section 8.2.2, then $U_M$ satisfies IRC.*

A proof of Theorem 3 is given in Appendix D.

## 8.3 Verifying WCFI-RW

In the remainder of this chapter, we describe an automatic, static verifier for proving that a user-provided $U_M$ satisfies the WCFI-RW property. Verifying such a property at machine code level brings up scalability concerns. Our benchmarks consist of enclave programs that are upwards of 100 KBs in binary size, and therefore whole-program analyses would be challenging to scale. Intra-procedural analysis, on the other hand, can produce too many false alarms due to missing assumptions on the caller-produced inputs and state. For instance, the caller may pass to its callee a pointer to some heap allocated structure, which the callee is expected to modify. Without any preconditions on the pointer, a modular verifier might claim that the callee writes to non-enclave memory, or corrupts a return address, etc.

Instead of verifying WCFI-RW of arbitrary machine code, our solution is to generate machine code using a compiler that emits runtime checks to enforce WCFI-RW, and automatically verify that the compiler has not missed any check. Our compiler emits runtime

checks that enforce that unconstrained pointers (e.g., from inputs) are not used to corrupt critical regions (e.g., return addresses on the stack), write to non-enclave memory, or jump to arbitrary code, etc. As our experiments show, the presence of these checks eliminates the unconstrained verification environments described above. Consequently, most verification conditions (VCs) that we generate can be discharged easily. Even in cases where the compiler eliminates checks for efficiency, the compiler does not perform any inter-procedural optimization, and we demonstrate that a modular verifier can prove that eliminating the check is safe.

### 8.3.1 Runtime Checks

We use the compiler to 1) prepend checks on `store` instructions to protect return addresses in the stack, 2) prepend checks on `store` instructions to prevent writes to non-enclave memory, and 3) prepend checks on indirect `call` and `jmp` instructions to enforce valid jump targets. We also use the processor's page-level access checks for efficiently preventing reads and writes to $L$'s memory by code in $U_M$.

**Runtime Checks on Stores:** To enforce that writes through pointers do not corrupt return addresses on the stack, the compiler maintains a bitmap (see `write_bitmap` in Figure 8.3) to record which areas in $U$'s memory are writable, while maintaining the invariant that a return address is never marked writable. The `write_bitmap` maps every 8-byte slot of $U$'s memory to one bit, which is set to one when those 8 bytes are writable. The bitmap is updated at runtime, typically to mark address-taken local variables and `malloc`-returned regions as writable, and to reset the bitmap at procedure exits or calls to `free`. For instance, if a caller expects a callee to populate the results in a stack-allocated local variable, the caller must mark the addresses of that local variable as writable before invoking the callee. A `store` instruction is prepended with an instruction sequence that reads the `write_bitmap` and terminates the enclave program if the corresponding bit is zero (see instructions from L1 to L2 in Figure 8.3). This check on `store` can enable stronger properties than backward edge CFI in that it also prevents many forms of memory corruptions. While this gives us stronger security guarantees at runtime, we only require a weak form of CFI for proving WCFI-RW.

In addition, the compiler prepends `store` with range checks that prevent writes outside the enclave memory (see instructions from L0 to L1 in Figure 8.3).

Finally, we use the processor's paging instructions to revoke write permissions on $L$'s memory while $U_M$ executes, and to reinstate write permissions following an API call to $L$ — we also use page permissions to make code pages and the `call_bitmap` non-writable at all times. The TAP guarantees that the processor respects the page permissions of SIR memory — in the case of Intel SGX 2.0, the processor provides the `emodpr` instruction to change page permissions of enclave's memory. The TAP processor performs the page-level access checks without any noticeable performance overhead. Note that we cannot use the page permissions

to prevent writes to non-enclave memory because the adversary $M$ controls the page tables for non-enclave memory.

**Runtime Checks on Loads:** WCFI-RW mandates that $U_M$ does not `load` from $L$'s memory. This ensures that $U_M$ never reads secrets such as the secure channel's cryptographic keys, which is necessary because strong encryption properties no longer hold if the key itself is used as plain text. To that end, $L$ disables read access to its memory by setting the appropriate bits in the page tables. On each API call, $L$ first sets the page permissions to allow access to its own memory, and then resets it before returning back to $U$.

On a side note, although we would like $U_M$ to only read enclave memory (and use `recv` to fetch inputs from non-SIR memory), we avoid introducing range checks for two reasons: 1) WCFI-RW does not require this guarantee, and 2) loads are frequent, and the range checks incur significant additional performance penalty.

**Runtime Checks on Indirect Control Transfers:** The compiler maintains a separate bitmap (see `call_bitmap` in Figure 8.3) that records the entry points of procedures in $U$ and APIs of $L$. The `call_bitmap` maps every 16-byte slot of $U$'s memory to one bit, and the compiler accordingly places each procedure's entry point in code at a 16-byte aligned address. The compiler prepends indirect calls with an instruction sequence that reads the bit within `call_bitmap` corresponding to the target address, and terminates the enclave program if that bit is zero. Indirect jumps to within the procedure are also checked to prevent jumps to the middle of x64 instructions, which can lead to control-flow hijacks.

The reader may question our choice of runtime checks, as one could simply instrument instructions implementing the validity checks on the corresponding transitions in the WCFI-RW monitor $\mathcal{M}$ (from Definition 6). However, this would require us to build a provably correct implementation of a *shadow stack* within enclave memory, and use the shadow stack during `call` and `ret` instructions to check that the processor uses the same return address as the one pushed by the matching `call` instruction. However, it is non-trivial to protect the shadow stack from code running at the same privilege level — doing so might require the very techniques that we use in our runtime checks.

### 8.3.2 Static Verifier for WCFI-RW

We present a modular and fully automatic program verifier, called /Confidential, for checking that the compiler-generated machine code satisfies the WCFI-RW property. Since runtime checks incur a performance penalty, the compiler omits those checks that it considers to be redundant. For instance, the compiler eliminates checks on writes to local scalar variables (and therefore does not need to make them writable in the `write_bitmap`) and to variables whose addresses are statically known. The compiler also tries to hoist checks out of loops whenever possible. These optimizations do not compromise WCFI-RW, and /Confidential proves them safe so as to avoid trusting the compiler. Since we verify

WCFI-RW at the machine code level, we do not need to trust the implementation of these optimizations.

/Confidential is based on a set of proof obligations for verifying that the output machine code (modeled as $U_M$) satisfies WCFI-RW. It modularly verifies each procedure in isolation and is still able to prove WCFI-RW for the entire program — this soundness guarantee is formalized as a theorem that we present later in this section. It is important to note that modular verification is possible because the compiler does not perform any global analysis to optimize away the runtime checks. /Confidential generates proof obligations for each `store`, `load`, `call`, ret, `jmp`, and rsp update in the procedure. While generating proof obligations, /Confidential does not distinguish instructions based on whether they originated from $U$'s source code or the runtime checks, since the compiler is untrusted. These proof obligations are implemented by instrumenting $U_M$ with static assertions, which are discharged automatically using an SMT solver by the process of VC generation. We present the instrumentation rules in Table 8.1, and describe them below.

| Instr $i$ | Instrumented Instr $I(i)$ |
|---|---|
| `call` $e$ | `assert` $\mathsf{policy}(e) \wedge (\forall i.\ (\mathsf{AddrInStack}(i) \wedge i < \mathsf{rsp}) \Rightarrow \neg\mathsf{writable}(\mathtt{mem}, i)) \wedge (\mathsf{rsp} \leq \mathsf{old}(\mathsf{rsp}) - 32)$;<br>`call` $e$ |
| $\mathtt{store}_{\mathsf{n}}(e_a, e_d)$ | `assert` $(\bigvee_i^{\{e_a, e_a+n-1\}} (\mathsf{AddrInStack}(i) \wedge i \geq \mathsf{old}(\mathsf{rsp}) \wedge \neg(\mathsf{old}(\mathsf{rsp}) + 8 \leq i < \mathsf{old}(\mathsf{rsp}) + 40)))) \Rightarrow \mathsf{writable}(\mathtt{mem}, e_a)$;<br>`assert` $(\bigwedge_i^{\{e_a, \ldots, e_a+n-1\}} (\mathsf{AddrInBitmap}(i) \Rightarrow (\mathsf{b}(\mathtt{mem}, i, e_d[8 * (i + 1 - e_a) : 8 * (i - e_a)]) < \mathsf{old}(\mathsf{rsp}) - 8)))$;<br>`assert` $\mathtt{evrange}(e_a) \wedge \mathtt{evrange}(e_a + n - 1)$;<br>$\mathtt{store}_{\mathsf{n}}(e_a, e_d)$ |
| rsp $:= e$ | `assert` $(e[3:0] = 000 \wedge e \leq \mathsf{old}(\mathsf{rsp}))$;<br>rsp $:= e$ |
| ret | `assert` $(\mathsf{rsp} = \mathsf{old}(\mathsf{rsp})) \wedge (\forall i.\ (\mathsf{AddrInStack(i)} \wedge i < \mathsf{old}(\mathsf{rsp})) \Rightarrow \neg\mathsf{writable}(\mathtt{mem}, i))$;<br>ret |
| `jmp` $e$ | `assert` $(\mathsf{start}(p) \leq e < \mathsf{end}(p)) \Rightarrow \mathsf{legal}(e)$;<br>`assert` $\neg(\mathsf{start}(p) \leq e < \mathsf{end}(p)) \Rightarrow (\mathsf{rsp} = \mathsf{old}(\mathsf{rsp}) \wedge \mathsf{policy}(e) \wedge (\forall i.\ (\mathsf{AddrInStack}(i) \wedge i < \mathsf{rsp}) \Rightarrow \neg\mathsf{writable}(\mathtt{mem}, i))$;<br>`jmp` $e$ |

*Table 8.1*: Instrumentation rules for modularly verifying WCFI-RW

The instrumentation rules use the following functions, which are defined for a given $U$:

- $\mathsf{policy}(e)$ is true iff address $e$ is the starting address of a procedure in $U$ or an API entrypoint of $L$. This predicate is consistent with the state of `call_bitmap`, which remains constant throughout the enclave's execution.
- $\mathsf{writable}(\mathtt{mem}, e)$ is true iff the bit corresponding to address `e` is set to one in the `write_bitmap` region of `mem`.

- $\mathsf{b}(\texttt{mem}, e_a, e_d)$ is a partial function (only defined for values of $e_a$ for which $\mathsf{AddrInBitmap}(e_a)$ holds) that returns the largest address that is marked writable as a result of executing $\texttt{store}(e_a, e_d)$
- $\mathsf{start}(p)$ returns the starting address of procedure $p$
- $\mathsf{end}(p)$ returns the ending address of procedure $p$
- $\mathsf{legal}(e)$ is true for any $e$ that is the starting address of an instruction in $U$ — we need this predicate because x64 instructions have variable lengths.
- $\mathsf{old}(\mathsf{rsp})$ is the value of $\mathsf{rsp}$ at procedure entry, and is modeled as a symbolic variable because the procedure may be called at an arbitrary depth in the call stack.

Functions policy, start, and end are defined by parsing the executable (DLL format) produced by the compiler. legal is defined by disassembling the executable code, which is a precursor to the formal modeling step that produces $U_M$. Since the memory layout and code pages remain constant throughout execution, these functions are defined once for a given $U$ and are independent of the current state. Functions writable and b are evaluated on the contents of `write_bitmap` within `mem`, and their definition involves a `load` from `mem` and several bitvector operations. We also recall predicates AddrInStack, AddrInBitmap, AddrInL, and `evrange` from section 8.2, which are used to define various regions in the SIR's memory (see Figure 8.3).

**Static Assertions on Calls:** On each statement of the type `call` $e$, we assert that 1) the target address $e$ is either a procedure in $U$ or an API entrypoint in $L$, 2) all addresses in the callee's stack frame are initially unwritable, and 3) the caller follows the x64 calling convention by allocating 32 bytes of scratch space for use by the callee.

**Static Assertions on Stores:** $U_M$ may invoke $\texttt{store}_\mathsf{n}(e_a, e_d)$ on an arbitrary virtual address $e_a$ with arbitrary data $e_d$. Hence, we must argue that the proof obligations prevent all `store` instructions that violate WCFI-RW. We case split this safety argument for each memory region in the virtual address space (Figure 8.3). The `call_bitmap`, the code pages, and $L$'s memory are marked non-writable in the page tables — `store` to these areas results in an exception, followed by termination. Within $U$'s memory, /CONFIDENTIAL treats all writes to the heap and globals as safe because WCFI-RW does not require any invariants on their state — while the heap and global area may store code and data pointers, /CONFIDENTIAL instruments the necessary assertions on indirect control transfers and dereferences, respectively. We are left with potential stores to $U$'s stack, `write_bitmap`, and non-SIR memory, and their proof obligations are:

- $\mathsf{AddrInStack}(e_a)$: if $e_a$ is an address in a caller's stack frame but not in the 32-byte scratch space (which is addressed from $\mathsf{old}(\mathsf{rsp}) + 8$ to $\mathsf{old}(\mathsf{rsp}) + 40$), then $e_a$ must be

marked by the `write_bitmap` as writable. On the other hand, $U_M$ is allowed to write arbitrary values to the current stack frame or the 32-byte scratch space.

- AddrInBitmap($e_a$): only addresses in the current stack frame can be made writable. It suffices to check that the largest address whose write permission is being toggled is below old(rsp). Note that unaligned stores may change two words at once. Since the instrumentation code only checks the write permissions of the first word (for performance reasons), we further restrict the largest address to be below old(rsp) − 8 to account for unaligned stores of up to 8 bytes.
- `evrange`($e_a$): WCFI-RW mandates that $U_M$ does not `store` to non-enclave memory, for which /CONFIDENTIAL generates a proof obligation: `assert evrange`($e_a$).

**Static Assertions on Assignments to rsp:** For each statement of the type rsp := $e$, we check that the new stack pointer $e$ 1) is 8-byte aligned, 2) does not point to a caller's stack frame (i.e., must not be greater than the old rsp). The constraint that the rsp never points to a caller's stack frame is necessary for modular verification. We use a guard page (i.e., a page without read or write page permissions) to protect against stack overflows — in the case where the procedure needs stack space larger than a page, we check that the compiler introduces a dummy load that is guaranteed to hit the guard page and cause an exception, thus preventing the procedure from writing past the guard page.

**Static Assertions on Returns:** For each ret statement, we check that 1) rsp has been restored to its original value, and 2) the procedure has reset the `write_bitmap` so that all addresses in the current stack frame are unwritable.

**Static Assertions on Jumps:** A `jmp` is safe if it either targets a legal address within the current procedure $p$ (i.e., not in the middle of an instruction), or the start of a procedure (often used for performing tail calls). In the case of `jmp` to a procedure, we check the same properties as a `call` instruction, except that rsp is restored to its original value.

**Syntactic Check for TAP primitives:** Code in $U_M$ runs at the same privilege level as $L$, and hence may invoke TAP's instructions to override page permissions (such as emodpr in SGX 2.0). Since $L$ performs all the cryptographic operations, and implements all security critical logic on behalf of $U$, we require that $U$ not invoke any of TAP's primitives, and only permit $L$ to invoke them. We simply check for the presence of such instructions (e.g. any SGX instruction) in $U$, which is captured by a regular expression on the disassembled machine code.

| Instr $i$ | Instrumented Instr $I(i)$ |
|---|---|
| $\texttt{store}_\mathsf{n}$ $(e_a, e_d)$ | $\texttt{assert}\ (\bigvee_i^{\{e_a, e_a+n-1\}}\ (\mathsf{AddrInStack}(i) \wedge i \geq \mathsf{old}(\mathsf{rsp}) \wedge \neg(\mathsf{old}(\mathsf{rsp}) + 8 \leq i < \mathsf{old}(\mathsf{rsp}) + 40))) \Rightarrow$<br>$\quad \mathsf{writable}(\texttt{mem}, e_a);$<br>$\texttt{assert}\ \bigwedge_i^{\{e_a, \ldots, e_a+n-1\}}\ (\mathsf{AddrInBitmap}(i) \Rightarrow$<br>$\quad (\mathsf{old}(\mathsf{rsp}) - \mathsf{estimate} \leq \mathsf{b}(\texttt{mem}, i, e_d[8 * (i + 1 - e_a) : 8 * (i - e_a)]) < \mathsf{old}(\mathsf{rsp}) - 8));$<br>$\texttt{assert}\ \texttt{evrange}(e_a) \wedge \texttt{evrange}(e_a + n - 1);$<br>$\texttt{store}_\mathsf{n}(e_a, e_d)$ |
| $\texttt{call}\ \ e$ | $\texttt{assert}\ \mathsf{policy}(e)\ \wedge\ (\mathsf{rsp} \leq \mathsf{old}(\mathsf{rsp}) - 32)\ \wedge\ (\mathsf{rsp} \leq \mathsf{old}(\mathsf{rsp}) - \mathsf{estimate});$<br>$\texttt{call}\ \ e$ |
| ret | $\texttt{assert}\ (\mathsf{rsp} = \mathsf{old}(\mathsf{rsp}))\ \wedge\ (\forall i.\ (i < \mathsf{old}(\mathsf{rsp}) \wedge i \geq \mathsf{old}(\mathsf{rsp}) - \mathsf{estimate}) \Rightarrow \neg\mathsf{writable}(\texttt{mem}, i));$<br>ret |

*Table 8.2*: Optimized instrumentation rules for `store`, `call`, and ret statements

### 8.3.3 Optimization to the Proof Obligations

If we can estimate the stack size needed for a procedure, then we can optimize the proof obligations for `store`, ret, and `call` statements (see Table 8.2). The modifications are:

- `store`: further assert that updating the `write_bitmap` does not mark any address below the estimated stack space to be writable.
- `call`: further assert that the current rsp is not within the estimated stack space (which would otherwise falsify the callee's precondition that the stack space is non-writable). The modified assertion on `store` also allows us to omit the proof obligation that all addresses below the current rsp are non-writable (prior to the `call`).
- ret: now asserts non-writability of only the addresses in the estimated stack space, instead of all addresses below the old(rsp). Since the range of addresses is bounded, we instantiate the $\forall$ quantifier, and help the SMT solver to eliminate hundreds of timeouts in our experiments.

Although the optimization is sound for any positive value of estimate, we are able to compute a precise value for all of our benchmarks. The estimate is computed by aggregating all the stack subtractions and checking that there is no assignment to rsp within a loop. If rsp is assigned within a loop body, then the optimization is disabled. Furthermore, this optimization may lead to false positives in rare cases of safe programs, in which case we also try verifying using the unoptimized implementation. For instance, this may happen if a procedure decrements rsp after making a procedure `call`, but we have not encountered such cases in our evaluation.

### 8.3.4 Soundness

The following theorem states that our proof obligations imply WCFI-RW.

**Theorem 4** *(Soundness of $I$) Let $p$ be a procedure in $U_M$, and $I(p)$ be procedure $p$ instrumented with the assertions given in Table 8.1 (or using the optimizations in Table 8.2). If for each $p$ in $U_M$, $I(p)$ is safe (i.e., no trace of $I(p)$ violates an assertion), then $U_M$ satisfies WCFI-RW.*

A proof of Theorem 4 is given in Appendix D.

## 8.4 Implementation

We develop a toolchain for building IRC-preserving enclaves. The developer first compiles $U$'s code (written in C/C++) using the compiler [107], to insert checks on indirect control-flow transfers, and checks on stores to prevent tampering of return addresses and to prevent stores from exceeding the enclave region — the instrumentation also protects against several classes of memory corruption errors, but we do not leverage these guarantees for proving IRC.

/CONFIDENTIAL takes as input a DLL with $U$'s code, and an implementation of $L$ that provides the expected guarantees (defined in Section 8.2). First, /CONFIDENTIAL parses the DLL to extract all the procedures. Next, for each procedure, /CONFIDENTIAL invokes the Binary Analysis Platform (BAP [31]) to lift the x64 instructions to instructions in our language (Figure 3.2), which are then modeled straightforwardly in BoogiePL [14]. The only caveat is that indirect `jmp` and `call` instructions require some preprocessing before modeling them in BoogiePL (since the control flow must be statically known before generating verification conditions), and we discuss this detail later in this section.

Next, for proving WCFI-RW, /CONFIDENTIAL instruments each procedure with `assert` statements as given in Table 8.1 and Table 8.2. The Boogie verifier [14] generates VCs and automatically discharges them using the Z3 SMT solver [41]. If all `assert` statements in all procedures are valid, then by Theorem 3 and Theorem 4, $U_M$ satisfies IRC. /CONFIDENTIAL checks the validity of each `assert` in parallel, which in combination with the modular analysis, allows /CONFIDENTIAL to scale to realistic enclave programs.

**Modeling Procedure Calls** Since the analysis is modular, /CONFIDENTIAL replaces each procedure call by a havoc to the machine state in lieu of specific procedure summaries. The havoc is performed by assigning fresh, symbolic values to volatile registers and CPU flags, and assigning a fresh, symbolic memory (called new_mem below) which is subject to certain constraints as shown below. We encode the constrained havoc to machine state using the following statements in order, which are instrumented after the `call` statement in $U_{\mathcal{H}}$.

- ▷ `assume` $\forall i.\ (\mathsf{AddrInStack}(i) \ \wedge \ i < \mathsf{old}(\mathsf{rsp}) \ \wedge$
  $\neg\mathsf{writable}(\mathtt{mem}, i)) \Rightarrow \mathsf{load}_8(\mathtt{mem}, i) = \mathsf{load}_8(\mathsf{new\_mem}, i)$
  A callee procedure may have an unbounded number of store instructions that can modify any memory location marked writable in the `write_bitmap` — the havoc must

preserve the non-writable locations in the current stack frame because our instrumentation guarantees that a callee cannot change their writability (as opposed to the heap which may be made writable by calling `malloc`, and then modified).

- ▷ assume $\forall i$. AddrInStack($b(\texttt{mem}, i, 11111111)$)
  $\Rightarrow \mathsf{load}_1(\texttt{mem}, i) = \mathsf{load}_1(\texttt{new\_mem}, i)$
  Our instrumentation guarantees that a portion of the `write_bitmap` (specifically the part that controls stack addresses) is restored on ret, which validates this assumption.

- ▷ mem := new_mem
  This step assigns a new memory that is related to the old memory by the above `assume` statements.

- ▷ havoc rax, rcx, rdx, r8, r9, r10, r11;
  This step havocs all volatile registers (as defined by the Windows x64 calling convention) with fresh, symbolic values.

- ▷ havoc ZF, AF, OF, SF, CF, PF;
  The callee may cause arbitrary updates to the CPU flags, which is modeled by this havoc.

The constrained havoc above models an arbitrary $U$ procedure that has an unbounded number of instructions of any type — we prove this lemma within the proof of the soundness theorem 4. The constrained havoc (in the statements above) is followed by a jump to the next instruction, as computed during disassembly. This is sound because WCFI-RW guarantees that the callee uses the return address placed by the caller. There is a caveat that a call to $L$'s API is replaced by its contract (defined in section 8.2.2) in lieu of the constrained havoc defined above.

We also assume the following preconditions at the beginning of each procedure:

- ▷ $\forall i$. (AddrInStack($i$) $\wedge$ $i <$ old(rsp)) $\Rightarrow \neg$writable(mem, $i$)
  This assumption treats all addresses in the local stack frame as non-writable upon procedure entry. It is upon the procedure to explicitly update the `write_bitmap` to make parts of its stack frame writable. This precondition is sound since we also enforce it at all call sites.

- ▷ AddrInStack(old(rsp)) $\wedge$ old(rsp)$[3:0] = 000$
  We assume the initial stack pointer must be within the stack region and that it is 8-byte aligned. This precondition is sound because we enforce this property on every assignment to rsp.

**Modeling Indirect Control Transfers.** In order to use VC Generation and SMT solving, we need to statically approximate the set of jump targets for each register-based indirect jump. While we can use standard off-the-shelf value analysis, we observed that the compiler

idiomatically places a jump table in memory, which is indexed dynamically to compute the target. Correspondingly, our verifier determines the base address of the jump table and reads its contents to compute the set of potential jump targets; this step is not trusted. An indirect jump is then modeled as a "switch" over direct jump statements to the potential targets, with the default case being `assert` false. The presence of `assert` false allows the approximation step to be untrusted. Indirect calls are handled using a similar method as direct calls; we introduce a constrained havoc on writable memory, volatile registers, and all CPU flags.

**Modeling Havocs from $M$.** While our adversary model requires inserting $\mathsf{havoc}_{\neg\mathsf{evrange}}\ \mathtt{mem}$ before each statement in $U$, it is efficient and sound to do so only before `load` statements [112]. We havoc the result of a `load` statement if the address is a location in non-enclave memory; $\mathsf{reg} := \mathsf{load}_{\mathsf{n}}(e)$ is transformed to $\mathsf{if}\ (\mathtt{evrange}(e))\ \{\mathsf{reg} := \mathsf{load}_{\mathsf{n}}(e)\}\ \mathsf{else}\ \{\mathsf{havoc\ reg}\}$.

**Verifying Procedures with Loops.** /Confidential uses a candidate loop invariant that a portion of the `write_bitmap` (specifically the part that controls stack addresses) is preserved across loop iterations — we expect this invariant to hold because 1) the compiler tends to set the `write_bitmap` only in the procedure's prologue, which occurs before loop bodies, and 2) our proof obligations guarantee that callees preserve this portion of the `write_bitmap`. Empirically, we find that this loop invariant is sufficient for proving our assertions within loop bodies.

## 8.5 Evaluation

We evaluate /Confidential on several enclave programs that process sensitive data, and we summarize the results in Table 8.3 and Figure 8.4. We choose the three largest Map-Reduce examples from VC3 [107]: Revenue, IoVolumes, and UserUsage. UserUsage and IoVolumes processes sensitive resource usage data from a cloud platform. IoVolumes processes storage I/O statistics, and UserUsage aggregates the total execution time per user. Revenue reads a log file from a website and calculates the total ad revenue per IP address. Each of these benchmarks implement the mappers and reducers within the enclaves, and place large parts of the untrusted Hadoop stack outside the enclave boundary. Performance evaluation of these applications is described in [107], which reports that the average cost of the run-time checks is 15%. We also experiment with three SPEC CPU2006 benchmarks: bzip2, astar, and lbm.

As Table 8.3 and Figure 8.4 show, /Confidential is able to prove almost all assertions needed to check WCFI-RW in less than 20 seconds each, which demonstrates the potential in scaling our approach. We performed full verification of the lbm benchmark (i.e., no timeouts or false positives), which took roughly 3 hours of wall clock time. We also discovered few procedures across many benchmarks that have instructions that BAP [31] could not process, and we plan to experiment with alternative tools in future.

All benchmarks were compiled with the optimization level at -O2. All experiments were performed on a machine with 160GB RAM and 12 Intel Xeon E5-2440 cores running at 2.40GHz. As mentioned previously, /CONFIDENTIAL parallelizes the verification by spawning several instances of the Z3 SMT solver, where each instance is responsible for proving one of the instrumented static assertions.

| Benchmark | Code Size | Verified Asserts | Timed out Asserts | False Positives |
|---|---|---|---|---|
| UserUsage | 14 KB | 2125 | 2 | 4 |
| IoVolumes | 17 KB | 2391 | 2 | 0 |
| Revenue | 18 KB | 1534 | 3 | 0 |
| lbm | 38 KB | 1192 | 0 | 0 |
| astar | 115 KB | 6468 | 2 | 0 |
| bzip2 | 155 KB | 10287 | 36 | 0 |

*Table 8.3*: Summary of results.

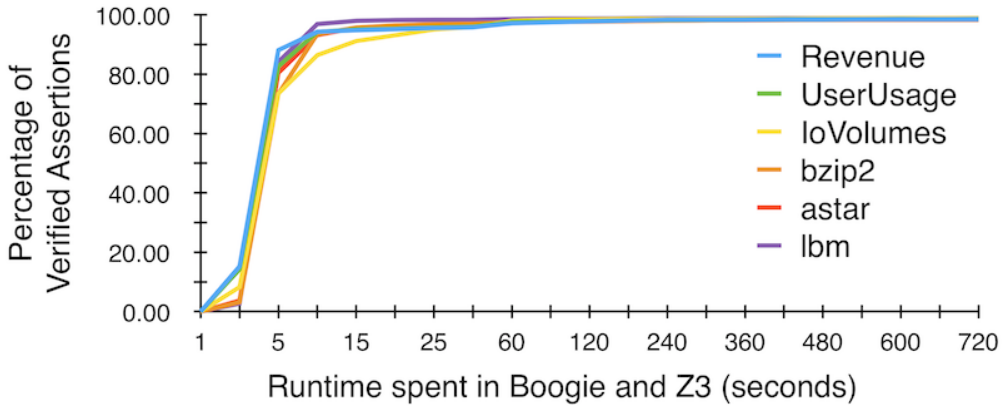



*Figure 8.4*: Summary of performance results

**False Positives.** We found four assertions that produced spurious counterexample traces, all within a single procedure of UserUsage. The violating procedure is a C++ constructor method, which writes the `vtable` pointer in the newly allocated object. Since the memory allocator terminates the enclave if it fails to allocate memory, the compiler optimizes away the range checks on the pointer returned by the memory allocator — this is the only observed instance of the compiler performing global optimization. Since /CONFIDENTIAL does not do any global analysis, it flags this method as unsafe. We circumvent this issue by disabling this optimization.

**Timeouts.** Prior to implementing the optimizations in section 5.3, we had experienced several hundred timeouts. After the optimizations, only roughly 0.2% of all assertions (across all benchmarks) do not verify within the 20 minute timeout, as shown in Table 8.3. The main source of complexity in the generated VC comes from the combination of quantifiers and multiple theories such as arrays and bitvectors that are typically hard for SMT solvers. Another reason is the presence of a few large procedures in the SPEC benchmarks — largest procedure has above 700 x64 instructions and 5200 BoogiePL statements — which generated large SMT formulas. These large procedures (considering those above 420 x64 instructions and 3500 BoogiePL statements) accounted for 31 (69%) timeouts. We found that all of these assertions are associated with `store` instructions, and they enforce that the `store` targets a writable region in memory — one feature of these large procedures is heavy use of the stack space in memory, potentially causing the SMT solver to reason heavily about aliasing in order to prove that the target address is marked non-writable. Ten (22%) of the timeouts are on assertions associated with ret instructions, where the solver struggled to prove that all locations in the current stack frame are made non-writable (even with the optimization in section 5.3) — unless the procedure explicitly resets the `write_bitmap` prior to the ret, the SMT solver must prove that none of the stores in the procedure are unsafe. The remainder of the timeouts, roughly 9%, were neither on the return instructions, nor in large procedures — we find that they are associated with `store` instructions, where the solver is able to prove that the `store` targets the `write_bitmap` but not whether the written value is safe.

We manually investigated the assertions that time out (by experimenting at the level of the BoogiePL program) and were able to prove some of them using additional invariants and abstractions, without requiring any specific knowledge of the benchmark or its source code. Although we machine check these proofs (using Boogie and Z3), we continue to count them as timeouts in Table 8.3, since our goal is to have a fully automatic verifier of WCFI-RW. For roughly half of the timeouts observed on ret instructions, we hypothesized intermediate lemmas (a few instructions prior to the ret) and simplified the VC by introducing a havoc to `mem` followed by an `assume` that is strong enough needed to prove the final assertion — we also prove that the intermediate lemma holds prior to the havoc, making this transformation sound. Specifically, for a stack address $a$, we either 1) havoc the `write_bitmap` if the procedure contains instructions to reset the `write_bitmap` corresponding to address $a$, and these instructions are sufficient to prove the final assertion $\neg\mathsf{writable}(\texttt{mem}, a)$, or 2) we introduce `assert` $\neg\mathsf{writable}(\texttt{mem}, a)$ at earlier points in the program, if the procedure does not make $a$ writable. This approach eliminates 6 of the 10 timeouts on ret instructions.

We also experimented with the 31 timeouts on `store` instructions within the large procedures. With the exception of 3 of these 31 timeouts, we were not able to get Z3 to prove the assertions, even after simplifying the VC with intermediate lemmas and havoc statements. These 3 assertions were at relatively shallow depths in the control flow graph of the procedure, where there are fewer loads and stores leading to the assertion. Finally, we tried the CVC4 [16] solver, but we did not succeed in eliminating any more timeouts.

Having performed this investigation, we are hopeful that with improving SMT solvers and better syntactic heuristics for simplifying the VCs, we will eliminate all timeouts.

## 8.6 Related Work

Confinement of programs to prevent information release has been studied for many years [67]. We are interested in achieving confinement in user programs, even with buggy or malicious privileged code. We use trusted processors [80, 9, 90, 69] to create isolated memory regions where we keep confidential application code and data, but our techniques are also applicable if isolation is provided by the hypervisor [60, 34, 127], or runtime checks [40]. Independently of the isolation provider, we need to reason about the application code to provide formal guarantees of confinement, which is the goal of /Confidential .

Our method to check for WCFI-RW draws inspiration from prior work to perform instrumentation in order to satisfy control-flow integrity (CFI [4, 89]) and software fault isolation (SFI [122, 78, 108]). Like SFI, we introduce run-time checks to constrain memory references. Our run-time checks are similar to the ones used in VC3 [107], but importantly we use the paging hardware to check reads, which is more efficient than relying on compiler instrumentation. Unlike VC3, we verify that our checks guarantee IRC. Native Client [130] also enforces a form of SFI, but its run-time checks for 64-bit Intel processors would require us to create enclaves with a minimum size of 100GB [108], which is not practical for our target environment (x86-64 CPUs with SGX extensions). This is because the enclave's address space must be statically configured and physically mapped by the CPU upon the enclave's creation, whereas the 64-bit Native Client scheme was implemented in a setting where the virtual address space can be large. Our run-time checks also enforce stronger security properties; for example, Native Client does not guarantee that calls return to the instruction immediately after the call. Native Client ultimately enforces a different policy: it aims to sandbox browser extensions and trusts the host OS, while we aim to isolate an application from a hostile host. This requires us to model a powerful, privileged adversary ($M$) while reasoning about the application's execution.

There have also been efforts to perform SFI with formally verified correctness guarantees. RockSalt [84] uses Coq to reason about an x86 processor model and guarantee SFI; it works for 32-bit x86 code, while our system works for the x86-64 ISA. ARMor [133] uses HOL to reason about ARM processor model and guarantee SFI. Native Client, XFI [49] and [132] include verifiers that work on machine code. Our verification scheme is different from these works since it uses different runtime checks (which provide stronger guarantees) and it supports aggressive compiler optimizations that remove redundant checks. We require more complex reasoning and thus use an SMT solver to build our verifier.

Unlike all of the above works, our ultimate goal is preserving confidentiality of a trusted application running in an untrusted and hostile host. Our specific definition of WCFI-RW, together with contracts we assume on the library methods guarantees IRC, which is the novel aspect of our work. We also prove that all the pieces (the compiler checks, the static verification, and the contracts on library methods) all combine together and guarantee IRC. Moat [112] has the same goal as our work, and the main difference is that Moat works for any code, and our work requires the application to perform all communications through a narrowly constrained interface. On the flip-side, Moat performs global analysis, tracks secrets

in a fine-grained manner, and is not scalable beyond programs containing few hundred x86 instructions. In contrast, /CONFIDENTIAL is modular, avoids fine-grained tracking of secrets, and hence scales to larger programs. As mentioned before, our notion of confidentiality does not prevent information leaks via side channels such as memory access patterns. This channel has been addressed in GhostRider [71], which presents a co-designed compiler and hardware (containing Oblivious RAM) for guaranteeing memory trace oblivious computation.

Translation validation (e.g., [96, 88, 119, 115]) is a set of techniques that attempt to prove that compiler optimizations did not change the semantics of the program given as input (after the optimizer run). The approach in /CONFIDENTIAL is similar in spirit to translation validation since we use an off-the-shelf, untrusted compiler and validate whether the code it produced satisfies the security properties we are interested in.

## 8.7 Summary

We presented a methodology for designing enclave programs, which enables certification of applications that need their code and data to remain confidential. Our methodology comprises enforcing the user code to communicate with the external world through a narrow interface, compiling the user code with a compiler that inserts run-time checks that aid verification, and linking it with a verified runtime that implements secure communication channels. We formalized the constraints on user code as Information Release Confinement (IRC), and presented a modular automatic verifier to check IRC. We believe that IRC, together with additional requirements on the implementation of the runtime, can guarantee a strong notion of confidentiality.

# Chapter 9

# Ensuring Page Access Obliviousness

While previous chapters on Moat and /Confidential protected enclaves from attacks that extract sensitive data via outputs (i.e., writes made by enclave to non-enclave memory), a sophisticated attacker can also recover the same data via any side channel exposed by the platform. For instance, on Intel SGX, the attacker can use the compromised OS to observe cache and page-level memory access patterns, which can divulge the same sensitive data that Moat and /Confidential were designed to protect. Recently, Xu et al. [125] demonstrated a side-channel exploit that extracts secrets from an enclave by observing its access pattern to code and data pages, which depends on sensitive data in a typical application. Therefore, in addition to protecting the enclave's outputs (e.g. by using tools such as Moat and /Confidential), the developer faces the burden of programming the enclaves correctly against privileged attacks that use side channels to infer an enclave's secrets. Practical defenses against such a privileged adversary is an open research problem. This chapter addresses the problem of defending against an adversary that can observe page-level access patterns of an enclave program — in other words, we show how to harden enclaves against a stronger threat model $MP$, defined in Section 4.2.

In Chapter 6, we formalized a confidentiality property, termed *page access obliviousness* (PAO), that asserts that the adversary's observation of page accesses must be independent of the enclave's secrets. Our key contribution is a method for compiling high-level source programs to machine code (containing x86-64 and TAP instructions) such that the machine code satisfies PAO, and a method for efficiently verifying that the enclave binary provably satisfies PAO. This is a first step towards making the developer blissfully unaware of the sophisticated attacks that can be mounted by a privileged adversary, allowing them to focus entirely on application logic.

In summary, this chapter makes the following novel contributions:

1. We present a toolchain, consisting of a type system and compiler, that automatically enforces PAO while compiling to machine code, and implements a stochastic optimization phase to reduce runtime and memory overheads.

2. To reduce our trusted computing base, we develop a separate verifier that analyzes the

compiled machine code to prove PAO. To further simplify the verifier, we design a typed assembly language, where the typing annotations accompany the machine instructions and reduce the verification task to efficient type checking.

3. We evaluate the toolchain on several machine learning algorithms and image processing routines, for which attacks have been demonstrated in [125].

The rest of this chapter has the following organization. Section 9.1 presents an overview of the PAO-enforcing compilation scheme, and the various challenges it must address. Section 9.2 discusses the internals of the compiler, and Section 9.3 discusses the machine code verifier which proves that the compiler output binary satisfies PAO. We evaluate the compiler and verifier tools in Section 9.4, and finally discuss related work in Section 9.5.

## 9.1 Overview

Page access Obliviousness (PAO), as defined in Definition 4, requires that the program exhibit the same ordering of accesses to memory pages regardless of the values of its secrets. We implement PAO enforcement within a compiler for EncLang, a general-purpose language for programming enclaves. The rationale for developing EncLang is two-fold: 1) memory accesses in mainstream languages (such as C, Rust) are determined by the compiler implementation, which offers us no control over the placement of objects and code in memory, thus hindering a static scheme for enforcing PAO — a dynamic scheme for LLVM bytecode [110] has been proposed, but with prohibitively high performance overheads; 2) because enclaves do not trust non-enclave software, they cannot use legacy software toolchains and thus present a rare opportunity for clean slate programming and verification. Our compiler accepts arbitrary programs in EncLang and *obliviates* the page accesses in the compiled x86-64 program, i.e., it satisfies PAO by making page access pattern independent of secrets. For instance, the compiler ensures that secret-dependent branches fetch instructions from the same (sequence of) pages in both branches. The compiler also performs stack allocation and lays out data structures in the enclave's heap so that a memory access via a secret address (e.g. array access with a secret index) generates a deterministic page access sequence in all executions. During this process, the compiler instruments dummy memory accesses or inserts padding space between objects in memory to *obliviate* the page accesses. For performance, the compiler also performs stochastic optimization, based on Markov Chain Monte Carlo sampling, to reduce the increase in code and data size of compiled programs. Although we implement these techniques in a compiler for EncLang, the ideas are generally applicable to mainstream languages. Moreover, we use EncLang to only program the enclave components of an application, which we empirically observe to be a small fragment of the entire application (e.g. map and reduce functions in VC3).

Next, we develop a separate verifier that proves that the output machine code satisfies PAO. The gains are two-fold: 1) the compiler is no longer trusted, thereby freeing us to implement aggressive optimizations without inadvertently sacrificing PAO, and 2) the verifier

is significantly simpler, and shrinks the size of our TCB, which now includes only the CPU hardware and our verifier. Furthermore, to alleviate the complexity of verifying arbitrary machine code, we engineer the compiler to supply (untrusted, but useful) hints to the verifier — we follow the typed assembly language paradigm of having the compiler output typing annotations along with the machine code, and create a set of simple typing judgments for efficient verification.

In the remainder of this section, we cover the key ideas behind our PAO-enforcing compilation and the machine code verification of PAO.

### 9.1.1 Threat Model

Our threat model was formalized earlier in Section 4.2, where we define the adversary $MP$ that observes both non-enclave memory and page-level accesses to memory (both enclave and non-enclave memory). In this subsection, for the reader's convenience, we recall the threat model, and the various operations and observations that the attacker $MP$ is allowed.

We assume a software adversary that has full control of all system software: OS, hypervisor, system management mode firmware, and BIOS. As a result of these privileges, the adversary has full control over non-enclave memory, I/O peripherals, disks, and network; it may record, replay, or modify network messages and disk contents. The adversary may force the CPU to transfer control from the enclave to the untrusted OS at any time during execution (by generating an interrupt, for example). Once the CPU transfers control to the adversary, the adversary may execute an arbitrary adversarial operations before transferring control back to the enclave.

To allow an OS autonomy over memory paging decisions, Intel SGX places the page tables under the OS control. For security, the CPU implements an inverse page table mapping to ensure that the OS cannot change the physical mapping for any address in enclave's region. However, at any point, the attacker may modify the page table entries to the effect of inducing a page fault on each enclave memory access (e.g. by clearing the valid bit). That being said, the OS' page fault handler only needs to know the accessed page (and not all bits of the address), hence the CPU clears the page offset bits from the faulting address (12 least significant bits for 4KB-sized pages) prior to delivering the page fault exception. This reveals the enclave's memory access patterns only at the page-level granularity. Recent attacks [125] on enclaves have extracted secrets via this channel, and preventing such leaks is the focus of this chapter.

Defenses against hardware attacks are out of scope for this paper. For instance, we assume that the adversary cannot physically attack the CPU package to extract secrets nor snoop on the hardware bus connecting the CPU and DRAM. The latter assumption prevents the adversary from learning access patterns at the byte-level granularity, which would necessitate a more sophisticated defense. We also don't defend against timing leaks, which may result from 1) timing of page accesses, 2) cache timing attacks which the attacker uses to infer access patterns at a cache-line granularity.

### 9.1.2 Challenges in Guaranteeing Page Access Obliviousness

PAO is a specific instantiation of a more general property of confidentiality based on non-interference, and it states that the adversary's observations of page accesses during enclave execution must be independent of the enclave's secrets. However, guaranteeing PAO for any application has its set of challenges. Consider a sample enclave in Figure 9.1, which evaluates a decision tree to classify an input instance — while we write this enclave in EncLang, its syntax and semantics is natural and similar to C, and we refer the reader to Section 9.2 for details on EncLang. The decision tree and the evaluate algorithm are known to the adversary, whereas the input instance and output decision must be kept confidential — to that end, we encrypt the output before writing to non-enclave memory. To classify the instance, the procedure traverses the tree (stored as a flattened array) starting from the root node, until it reaches a leaf node (which is any node with a non-zero value in the decision field); the evaluation uses an index variable to record the current node in the traversal. At each interior node, the procedure compares the value of a decision variable with a threshold value, and recurses on either the left or right subtree based on the outcome. Note that, for a large decision tree, the left and right children may lie on separate data pages. Furthermore, the code implementing the left and right traversal may lie on separate code pages. Therefore, the path taken through the tree reveals predicates that hold on the secret instance, which the attacker infers by monitoring the enclave's accesses to code and data pages. To remedy such leakages, amongst other measures, the enclave developer must ensure that the page accesses are independent of the path, a property entailed by PAO. Guaranteeing PAO for the evaluate procedure has the following challenges:

- In the case of an unbalanced decision tree, evaluate terminates after varying number of iterations (based on secret), and the attacker may infer the path length by counting the number of page accesses. In other words, each invocation of evaluate leaks at most $\log_2 k$ bits of secret, where $k$ is the height of our decision tree.
- We have a secret-dependent conditional statement (line 23). Monitoring the enclave's accesses to the code pages allows the attacker to infer which branch is taken, if any of the instructions implementing the if branch (line 25) is placed in a different page than the instructions implementing the else branch (line 27). Mainstream compilers often optimize for code size and performance, but make no effort to control the layout of instructions.
- We find several data accesses where the address depends on a secret value. For instance, in lines 22 - 27, the array access tree[index] computes a reference to a node within the tree, where index is a secret; this makes the address evaluate to a secret value. In the case that tree is stored across multiple pages — because the tree size is larger than a single page, or due to layout decisions made by the compiler — the attacker infers some bits of the secret index by monitoring the enclave's accesses to data pages.

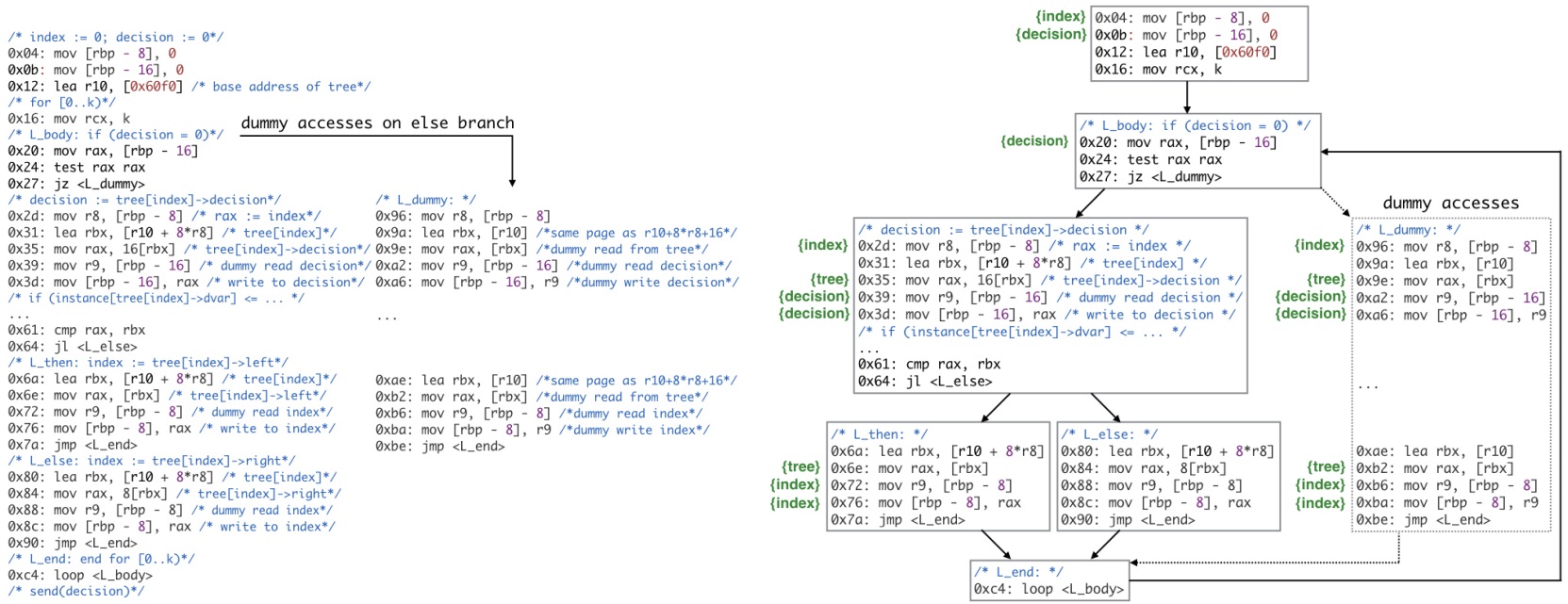
```
1  global tree:
2    array[2^k-1]
3      struct {
4        left : idx<2^k><public>, /* index of left subtree*/
5        right : idx<2^k><public>, /* index of right subtree*/
6        decision : uint64<public>, /* != 0 for leaf node */
7        dvar : idx<d><public>, /* decision variable */
8        threshold : uint64<public> /* threshold value */
9      };
10
11 void evaluate(instance : ref array[d]<secret> uint64 )
12 {
13   local decision: uint64<secret>; /* evaluation result */
14   local index: idx<2^k><secret>; /* values 0 to 2^k-1 */
15
16   index := 0;          /* start traversal at root */
17   decision := 0;       /* terminates when non 0 */
18
19   while (decision = 0) {
20   /* for (0..k) { */
21     if (decision = 0) {
22       decision := tree[index]->decision;
23       if (instance[tree[index]->dvar] <=
24           tree[index]->threshold) {
25         index := tree[index]->left; /* recurse left */
26       } else {
27         index := tree[index]->right; /* recurse right */
28       }
29     }
30   }
31   send(decision);
32 }
```

*Figure 9.1*: Decision tree evaluation illustrating some challenges in guaranteeing page access obliviousness.

### 9.1.3 Compilation for Page Access Obliviousness

We develop a compiler for producing PAO-satisfying x86-64 code from arbitrary EncLang programs. First, a type system (described in Section 9.2.5) flags violations where the enclave leaks secrets in ways that an automatic compiler cannot fix (without developer intervention): loops with secret-dependent condition, i.e., secret number of iterations, explicit leaks via assignment of secret values to public state, and implicit leaks via assignment to public state within a secret conditional branch — these typing restrictions are common amongst type systems for non-interference and side-channel mitigations. In the evaluate procedure in Figure 9.1, the developer replaces the secret-dependent loop condition (line 19) with a loop that executes for fixed number of iterations (line 20), thus trivially satisfying the typing rule that loop exit conditions must only depend on public values. The type checker finds no other violations.

The compiler (described in Section 9.2) then compiles to x86-64 code, listed in Figure 9.2(a), while also enforcing PAO by controlling the layout of data structures and in-




(a) Compiler safely lays out code and data pages, and instruments (b) Verifier constructs CFG and computes aliases for all memory accesses dummy accesses

*Figure 9.2*: Compiling and verifying evaluate to guarantee page access obliviousness

structions in memory — where necessary, it generates dummy page accesses to *obliviate* the access patterns, as discussed below. It takes the following necessary measures for our sample enclave.

Even with the fix on line 20, the enclave remains vulnerable — the program effectively stops computing and does not perform data accesses once the traversal reaches a leaf node (i.e., condition on line 21 evaluates to `false`), thus allowing the adversary to infer the path length by counting the accesses to the data pages. To correctly conceal this leakage, the compiler places dummy accesses in the `else` branch (corresponding to the `if` on line 21) to account for the imbalance in the tree. As seen in Figure 9.2(a), the dummy accesses are performed using instructions within the address range 0x96 to 0xbe, and they target the same sequence of pages as the two program paths in the input program — this is achieved via dummy reads from the same object, and by controlling the placement of objects.

Second, to prevent secrets from leaking via data accesses (e.g. tree[index], where the address depends on a secret), the compiler must either 1) layout the tree to fit entirely within a page (if possible), causing all accesses to the tree to target the same page, or 2) allow the tree to span multiple pages, and introduce dummy accesses to all pages except the page containing tree[index]. For a simpler presentation, the compilation in Figure 9.2 assumes that the tree object fits within a single page — Section 9.2.1 presents a general scheme that handles objects of arbitrary size. Despite this simplifying assumption, we must generate dummy accesses to tree in the `else` branch (corresponding to the `if` on line 21). A dummy read (e.g. instruction 0x9e, which mimics 0x35) is performed by fabricating an address within the tree. A dummy write (e.g. instruction at 0xa6, which mimics 0x3d) is performed by first issuing a dummy read (instruction 0xa2, which we compensate by adding instruction 0x39 in the original path), and then writing the read value back at the same address, thus preventing dummy accesses from modifying state.

Finally, To hide control decisions based on secret input (line 23), the compiler lays out the instructions from both branches onto the same page, when possible. Inevitably, to handle cases where the cumulative code within the branches of a secret conditional cannot be fit onto a single page, the compiler partitions code across pages such that the sequence of code page accesses is equivalent in the two branches — the compiler splits each branch into snippets, and maps snippets to pages such that the $n$th chunk of both branches have equal number of instructions and occupy the same page. Finally, the compiler instruments nop instructions to equalize the number of instructions (code accesses) in the two branches. Further details are presented in Section 9.2.3. For simplicity, we elide the nop instructions in Figure 9.2, and also manage to place all of the compiled machine code for evaluate within one page.

**Verifying Page Access Obliviousness** An enclave violates PAO if executions with different secret values produce different sequences of page accesses. We verify that the output machine code satisfies PAO, thus removing the compiler's implementation from the trusted computing base.

In this paper, we show that verifying PAO requires sound (but necessarily incomplete) algorithms for alias analysis and control flow analysis — in practice, we are able to implement

a simple, yet precise algorithm for these analyses because our compiler produces idiomatic code and supplies hints to the verifier, following the paradigm of typed assembly language. First, our verifier takes the enclave program as input and computes its control flow graph (CFG), as shown in Figure 9.2(b). Next, our verifier performs an alias analysis, annotating each memory access with a set of objects that the access may target. We show the aliases within curly braces in Figure 9.2(b); the verifier also annotates the aliases for the dummy accesses along the else branch (shown within the dotted box in the CFG), corresponding to the if in line 21. The verifier uses these analyses to prove PAO: for any pair of executions of the enclave binary (that only differ in secret values), the sequence of page accesses must be equivalent, where two accesses are equivalent if they target the same page and have same type (read / write / execute). The machine code in Figure 9.2 satisfies PAO trivially because the alias analysis computes only one object for each memory access, and all paths have equivalent sequence of accesses to code and data pages.

## 9.2 PAO-Enforcing Compilation

This section presents an algorithm for obliviating data accesses at the machine code level (Section 9.2.1), a stochastic optimization step for lowering runtime overheads of this defense (Section 9.2.2), and an algorithm for obliviating code accesses (Section 9.2.3). We also present EncLang (Section 9.2.4) and its compiler (Section 9.2.5, Section 9.2.6 ), which implements these algorithms to produce PAO-satisfying machine code.

### 9.2.1 Obliviating Data Accesses

Consider the following secret-dependent conditional branch:

```
if (s) {       /* s: bool<secret> */
  b[i] := a[k]; /* a: array[5000] uint8<public> */
} else {       /* b: array[10] uint8<secret> */
  c := 0;      /* c: uint64<secret> */
}
```

Consider two execution traces of this enclave, denoted $\pi_1$ and $\pi_2$, which have potentially different values of secret state variables. With different values of the secret s in executions $\pi_1$ and $\pi_2$ (from definition 3), the attacker observes different data accesses in $\pi_1$ and $\pi_2$:

- $\pi_1$ (if branch): $R[\![\mathsf{s}]\!]$, $R[\![\mathsf{a[k]}]\!]$, $W[\![\mathsf{b[i]}]\!]$
- $\pi_2$ (else branch): $R[\![\mathsf{s}]\!]$, $W[\![\mathsf{c}]\!]$

Here, $R[\![]\!]$ and $W[\![]\!]$ indicate read and write operation, respectively — we defer the treatment of code accesses (for fetching instructions) to section 9.2.3. A strawman PAO-enforcement scheme *obliviates* page accesses in the two branches by introducing dummy read ($\hat{R}[\![e]\!]$) and dummy write ($\hat{W}[\![e]\!]$) operations, which are guaranteed to 1) target the same page as the real read ($R[\![e]\!]$) and write ($W[\![e]\!]$) operations that they mimic, and 2) not cause any side-effect

| **Reference Type** | $R[\![e]\!]$: gets input $e$ in rax, and produces output in rax | $\hat{R}[\![e]\!]$: gets input $e$ in rax |
|---|---|---|
| $\Gamma \vdash_e si(e)$ : ref cell ... | mov rax [rax] | mov rax [rax] |
| $\Gamma \vdash_e si(e)$ : ref array ...<br><br>let s = ⌈ size(si(e)) / $2^p$ ⌉ + 1<br>denote the maximum number<br>of pages that si(e) can occupy | for k in [0..s − 1] {<br>  r11 := pg(si(e)) + $2^p$ ∗ k ≥ rax < pg(si(e)) + $2^p$ ∗ (k + 1)<br>  rcx := omove(r11, rax, pg(si(e) + $2^p$ ∗ k))<br>  mov rcx [rcx]<br>  rdx := omove(r11, rcx, rdx)<br>}<br>rax := rdx | for k in [0..s − 1] {<br>  rdx := pg(si(e)) + $2^p$ ∗ k<br>  mov rdx [rdx]<br>} |

| **Reference Type** | $W[\![e]\!]$: gets input $e$ in rax, and value in rbx | $\hat{W}[\![e]\!]$: gets input $e$ in rax |
|---|---|---|
| $\Gamma \vdash_e si(e)$ : ref cell ... | mov rdx [rax]<br>mov [rax] rbx | mov rdx [rax]<br>mov [rax] rdx |
| $\Gamma \vdash_e si(e)$ : ref array ...<br><br>let s = ⌈ size(si(e)) / $2^p$ ⌉ + 1<br>denote the maximum number<br>of pages that si(e) can occupy | for k in [0..s − 1] {<br>  r11 := pg(si(e) + $2^p$ ∗ k) ≥ rax < pg(si(e)) + $2^p$ ∗ (k + 1)<br>  rcx := omove(r11, rax, pg(si(e)) + $2^p$ ∗ k)<br>  mov r12 [rcx]<br>  rdx := omove(r11, rbx, r12)<br>  mov [rcx] rdx<br>} | for k in [0..s − 1] {<br>  rcx := pg(si(e)) + $2^p$ ∗ k<br>  mov rdx [rcx]<br>  mov [rcx] rdx<br>} |

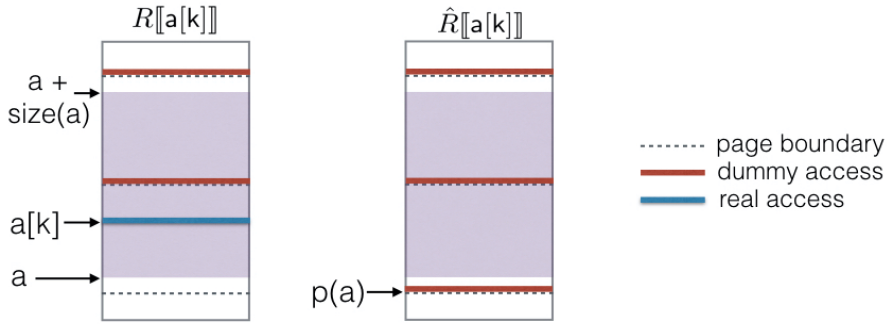



*Figure 9.3*: Implementation of oblivious read / write primitives. The omove primitive performs a conditional move using the cmovz instruction. pg(x) denotes the starting address of the page containing address x and is defined to be x & !($2^p$ − 1).

on the program's state. This approach results in the following data accesses along the two branches of the above program:

- $\pi_1$ (if branch): $R[\![\mathsf{s}]\!]$, $R[\![\mathsf{a[k]}]\!]$, $W[\![\mathsf{b[i]}]\!]$, $\hat{W}[\![c]\!]$
- $\pi_2$ (else branch): $R[\![\mathsf{s}]\!]$, $\hat{R}[\![\mathsf{a[k]}]\!]$, $\hat{W}[\![\mathsf{b[i]}]\!]$, $W[\![\mathsf{c}]\!]$

The implementation of dummy operations $\hat{R}[\![]\!]$ and $\hat{W}[\![]\!]$, which execute in a different code path than the real operations they mimic, must compute the same address or at least an address to the same page. This is trivial for scalar objects (e.g. variable c, which has a fixed location on the program stack). In the case where the object spans multiple pages (e.g. larger-than-page array a in $R[\![\mathsf{a[k]}]\!]$), we cannot statically identify a unique page targeted by a read / write operation — in general, the address may be computed using a secret (e.g. secret k in a[k]), and secrets may evolve to different values within the two branches of a secret-dependent conditional. Therefore, the dummy operation $\hat{R}[\![\mathsf{a[k]}]\!]$ must access each page that contains some part of the object, which also forces us to access the same pages to implement the real $R[\![\mathsf{a[k]}]\!]$ lest we violate PAO. In other words, the $R[\![]\!]$, $\hat{R}[\![]\!]$, $W[\![]\!]$, and $\hat{W}[\![]\!]$ operators — defined in Figure 9.3 and described below — may perform multiple memory accesses for each read / write operation.

As a first step, given a $R[\![e]\!]$ or $W[\![e]\!]$ operation to mimic, we perform a best-effort analysis to identify the object being targeted by the reference $e$, hereby called the *statically-identifiable* object. The statically-identifiable object, or $si(e)$, denotes a (contiguous) region of memory that is guaranteed to contain the address $e$, in all executions of the program. EncLang provides two constructs for computing references: the $\rightsquigarrow$ operator converts a reference to a struct into a reference to the named field, and the [.] operator returns a reference to the indexed array element — both operators have standard C-like semantics. We compute $si(e)$ syntactically by traversing struct accesses in $e$ until we arrive at a scalar or an array access — we stop the recursion at an array access to avoid performing range analysis, thereby allowing the computation of $si$ to be syntactic. For instance, $si(\mathsf{a[k]})$ is a; $si(\mathsf{w} \rightsquigarrow \mathsf{x[y} \rightsquigarrow \mathsf{z]})$ is $\mathsf{w} \rightsquigarrow \mathsf{x}$; $si(\mathsf{w} \rightsquigarrow \mathsf{x})$ is $\mathsf{w} \rightsquigarrow \mathsf{x}$; $si(\mathrm{s})$ is s. In general,

$$si(e) \doteq \begin{cases} id & \text{where } e \leftarrow \mathsf{id} \\ si(e_a) & \text{where } e \leftarrow e_a[e_i] \\ si(e_s) & \text{where } e \leftarrow e_s \rightsquigarrow \mathsf{id} \text{ and } e_s \text{ contains array access} \\ e & \text{where } e \leftarrow e_s \rightsquigarrow \mathsf{id} \text{ and } e_s \text{ has no array access} \end{cases}$$

Figure 9.3 defines the machine code implementation of the $R[\![e]\!]$, $W[\![e]\!]$, $\hat{R}[\![e]\!]$, and $\hat{W}[\![e]\!]$ operators. If $si(e)$ resolves to a scalar value (i.e., ref cell in EncLang), then both $R[\![e]\!]$ and $\hat{R}[\![e]\!]$ execute a single mov instruction to read the value. Else, we perform a linear scan over all pages that contain the array referenced by $si(e)$ (as illustrated in Figure 9.3). To do so, it suffices to know the size of the array referenced by $si(e)$, which we compute using the type inference in EncLang (Section 9.2.5) — the base address can be evaluated at runtime. Assuming the worst case layout of $si(e)$, where the object may span upto $\lceil \mathsf{size(si(e))} \,/\, 2^{\mathsf{p}} \rceil + 1$ pages, we perform the linear scan in $\hat{R}[\![e]\!]$ by issuing dummy loads from all these pages,

and perform the linear scan in $R[\![e]\!]$ by issuing a real load to the page containing address $e$ and dummy loads to the remaining $\lceil$ size(si(e)) / $2^{\mathsf{p}}$ $\rceil$ pages. To an attacker, a real load is indistinguishable from a dummy load — PAO is preserved by having equivalent number and type of page accesses in $R[\![e]\!]$ and $\hat{R}[\![e]\!]$. Observe that a dummy load targets the lowest address of a page (i.e., bottom p bits are 0), and discards the result of the load, whereas a real load targets the intended address $e$ and saves the result in rax. The implementation is similar for $W[\![e]\!]$ and $\hat{W}[\![e]\!]$. The linear scan uses dummy stores that target the lowest address of a page, and first loads a value from that address, and then stores that same value back. Meanwhile, a real store first loads a value from the target address into a dead register, and then stores the new value — to an attacker, a real store is indistinguishable from a dummy store. The implementation makes use of an oblivious move primitive called omove, which we borrow from [91]. Effectively, dst := omove(c, x, y) performs a conditional move without introducing a conditional branch — it moves both x and y into temporary registers, evaluates c, and uses the cmovz instruction to move either x (if c is true) or y (if c is false) into dst.

Note that the compiler only introduces dummy operations $\hat{R}[\![e]\!]$ and $\hat{W}[\![e]\!]$ within secret-dependent conditional branches. Code within public conditionals do not require any PAO-related defenses, as the program executes the same branch in both $\pi_1$ and $\pi_2$, and hence the attacker observes equivalent page accesses. Although, for simplicity, our example only contains one conditional statement, the algorithm handles arbitrary nesting of conditional statements. This section proposed a naive, but sound defense for enforcing PAO; the following section optimizes the program to remove unnecessary dummy accesses, without compromising soundness.

### 9.2.2 Stochastic Optimization of Dummy Accesses

So far, our strategy for obliviating data accesses assumed the worst-case layout of objects in memory. First, the $R[\![]\!]$ and $W[\![]\!]$ operators produce $\lceil$ size(si(e)) / $2^{\mathsf{p}}$ $\rceil + 1$ page accesses, accounting for the pathological case where even a two-byte object can span two pages — we observe this phenomenon in $W[\![\mathsf{b[i]}]\!]$, which needs only one page access if the 10-byte array b can be placed entirely within a page. Second, the strategy of introducing one dummy operator for each real operator in the other path assumes the worst case setting where all objects are placed in unique pages, which is rarely the case — we observe this phenomenon in the toy example from section 9.2.1, where both $\hat{W}[\![\mathsf{b[i]}]\!]$ and $\hat{W}[\![\mathsf{c}]\!]$ can be elided objects if b and c are placed in the same page. With this insight, we have an optimized sequence of data accesses:

- $\pi_1$ (if branch): $R[\![\mathsf{s}]\!]$, $R[\![\mathsf{a[k]}]\!]$, $W[\![\mathsf{b[i]}]\!]$
- $\pi_2$ (else branch): $R[\![\mathsf{s}]\!]$, $\hat{R}[\![\mathsf{a[k]}]\!]$, $W[\![\mathsf{c}]\!]$

We formulate this program transformation as a stochastic optimization problem, and solve it using Markov Chain Monte Carlo (MCMC) sampling, which quickly explores a large

search space of program rewrites. The cost function to optimize is the number of dummy page accesses. The MCMC sampling chooses amongst two candidate moves:

1. *Map objects to pages*: Randomly select a stack-allocated object and place it onto a random page (on the current stack frame) that has enough contiguous, free memory.

2. *Dummy accesses*: Randomly select a pair of $\hat{R}[\![]\!]$ (or $\hat{W}[\![]\!]$) operations and remove it from the program. Or, for ergodicity [106], create a pair of $\hat{R}[\![]\!]$ (or $\hat{W}[\![]\!]$) operations, each with a randomly chosen object as the argument.

These optimizations require the compiler to control the placement of local, stack-allocated objects. We assume no control over heap layout as it is performed by malloc, and the layout of globals is performed independently of all procedures — in principal, these can be optimized as well, but that would add unmanageable complexity to the compiler.

### 9.2.3 Obliviating Code Accesses

The branches of a secret-dependent conditional may contain unequal number of instructions, and hence produce unequal number of code page accesses in execution traces $\pi_1$ and $\pi_2$ — in fact, the reader may notice in Figure 9.3 that although our implementation of $R[\![e]\!]$ produces equal number of data accesses as $\hat{R}[\![e]\!]$, $R[\![e]\!]$ uses far more instructions. To obliviate the code accesses, the compiler instruments nop instructions on the same page as the instructions being mimicked — we apply this defense after the transformations in sections 9.2.1 and 9.2.2, ensuring that enough code accesses are interleaved with the data accesses to satisfy PAO. Note that this instrumentation is only applied to secret-dependent conditionals.

The nop instrumentation ensures that all paths within a secret-dependent conditional have equal number of x86-64 instructions, thus guaranteeing that equal number of code page accesses. However, this is not enough for PAO. Without consideration to code layout, instructions in different branches may get placed onto separate pages. This may happen if there is not enough space left on the current code page, or if the cumulative size of the branches is larger than a page. In such cases, we split each branch into a sequence of snippets, map snippets onto pages such that the $n$th snippet of both branches has equal number of instructions and occupies the same page, and stitch the snippets together using unconditional jumps.

### 9.2.4 The EncLang Language

Our goal is to empower the developer with a PAO-enforcing compiler and verifier toolchain, which implements the defenses in sections 9.2.1-9.2.3. Modifying off-the-shelf compilers for mainstream languages would require significant overhaul. We must at the very least: (1) add security types to identify secret branch conditions — in the case of C/C++, the type

system is also unable to strictly enforce type and memory safety, making it exceedingly difficult to statically reason about aliasing, control flow, and object sizes (e.g. evaluating *si* in 9.2.1); (2) extend the operational semantics to define memory accesses for each expression and statement in the language and enforce these semantics in all phases of the compilation — mainstream languages (e.g. C, Rust) leave these details to a compiler; (3) extend the compiler implementation with techniques proposed in this paper: issue dummy accesses, place nop instructions, arrange objects in data pages, arrange instructions in code pages, etc. — mainstream compilers don't offer this level of control over the code generation. Therefore, we implement our PAO enforcement within a compiler for a new language, called EncLang, that is mostly inspired from Ivory [48], Cyclone [62] and the work by Jones [45]. Note that EncLang is used to program the enclave components, whereas the rest of the application is developed using legacy toolchains. Figure 9.4 presents the syntax, and the rest of this section discusses the key features of EncLang.

Establishing memory safety is paramount for enforcing PAO — a memory safety violation, such as a control flow attack, may perform arbitrary, malicious computation within enclaves, making it infeasible to give any meaningful security guarantee. To that end, the EncLang restricts how references to objects are computed and stored. The language forbids out-of-bounds access and nullable pointers. The developer may allocate memory for either primitive types (cell $\beta$, where $\beta$ is a machine word type) or aggregate structures of type struct and array. The array size must be declared statically, and the array is indexed using a special integer type idx⟨k⟩, which represents values from 0 to k − 1. All allocations are either local (in the current procedure's stack frame), global, or on the heap (managed with malloc and free). Though we intend to relax this restriction in future work, we force the programmer to specify the type of the object (which also specifies its size) as an argument to malloc, in lieu of introducing runtime bounds checks. Allocations (via local, global, and malloc) bind the object's name to a reference that points to the base of the object. The $\rightsquigarrow$ operator converts a reference to a struct into a reference to the named field, and the [.] operator returns a reference to the indexed array element (if the type checker can prove that the index's type cannot allow an out-of-bounds access). These are the only two ways of computing references to objects in memory (pointer arithmetic is disallowed), and we also prevent references from being stored in memory to keep alias analysis simple yet precise (though we intend to relax this in future). Once a reference to a cell is obtained, deref and store is used to read and write a machine word.

A program may save intermediate results on the local stack using let statements and local declarations. A procedure (including malloc, free) is invoked using a call statement, and execution terminates within the procedure at the ret statement. Our type checker guarantees that references to stack allocated objects are not returned from a procedure, thus eliminating the possibility of a dangling pointer.

A standard feature of information flow type systems is a secrecy type, which has two values: public ($\bot$) or secret ($\top$). EncLang allows both values (i.e., cell) and references to have type $\bot$ or $\top$ — a secret reference indicates that the pointer is computed using secrets e.g. a[i], where i is secret. The type checker uses these types to identify illegal information flows

(e.g. storing a secret value in a public state variable), and the PAO enforcement algorithm (Section 9.2.1-Section 9.2.3) uses these types to identify the secret-dependent branching conditions.

**Advantages of using EncLang for PAO** The compiler can perform simple, yet precise, alias analysis to determine the set of objects that are potentially targeted by a store or deref.This is because the language provides limited constructs ($\rightsquigarrow$ and [.]) to compute pointers, and such pointers are always computed in a small number of registers and never saved in memory. In principle, we can relax these language restrictions, and develop sophisticated alias analysis, value analysis, etc. as part of the PAO-enforcing compilation. However, we find these challenges to be orthogonal to the key problems addressed in this work.

***Const***

$$
\begin{array}{lll}
n & ::= & \mathbb{N} \\
c & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{0bv8} \mid \ldots \mid \mathsf{18446744073709551615bv64}
\end{array}
$$

***Vars***

$$
\begin{array}{lll}
v & ::= & \mathsf{id}
\end{array}
$$

***Ops***

$$
\begin{array}{lll}
\otimes & ::= & + \mid \gg \mid \ll \mid * \mid = \mid < \mid \wedge \mid \vee \mid \neg \mid \ldots
\end{array}
$$

***Region***

$$
\begin{array}{lll}
r & ::= & \mathsf{local} \mid \mathsf{global} \mid \mathsf{heap}
\end{array}
$$

***Types***

$$
\begin{array}{lll}
l & ::= & \mathsf{public}(\bot) \mid \mathsf{secret}(\top) \\
\tau & ::= & \mathsf{ref}\langle l \rangle\ r\ \alpha \mid \beta \\
\beta & ::= & \mathsf{bool}\langle l \rangle \mid \mathsf{sint}\langle n, l \rangle \mid \mathsf{uint}\langle n, l \rangle \mid \mathsf{idx}\langle n, l \rangle \\
\alpha & ::= & \mathsf{array}\ n\ \alpha \mid \mathsf{struct}\{\mathsf{id} : \alpha, \ldots, \mathsf{id} : \alpha\} \mid \mathsf{cell}\ \beta\langle l \rangle
\end{array}
$$

***Expr***

$$
\begin{array}{lll}
e & ::= & v \mid c \mid e \otimes e \mid e \rightsquigarrow \mathsf{id} \mid e[e] \mid \mathsf{deref}\ e
\end{array}
$$

***Stmt***

$$
\begin{array}{lll}
s & ::= & s\ ;\ s \mid \mathsf{local}\ v : \alpha\ \mathsf{in}\ s \mid \mathsf{let}\ v = e\ \mathsf{in}\ s \mid \mathsf{store}\ e\ e \mid \mathsf{ret}\ e \\
 & & \mathtt{call}\ v = \mathsf{id}(e, \ldots)\ \mathsf{in}\ s \mid \mathsf{if}\ (e)\ \{s\}\ \mathsf{else}\ \{s\} \mid \mathsf{while}(e)\ \{s\}
\end{array}
$$

***Prog***

$$
\begin{array}{lll}
p & ::= & \mathsf{proc}\ \mathsf{id}(v : \tau, \ldots, v : \tau \rightarrow \tau)\{s\} \mid \mathsf{var}\ v : \alpha \mid p; \ldots; p
\end{array}
$$

*Figure 9.4*: EncLang syntax

### 9.2.5 Type Checking of EncLang programs

Figure 9.5 presents a type system for EncLang. The type checker flags information flow violations that a compiler cannot automatically repair e.g. assigning secrets to public state variables. A typing judgment for a well-typed statement $s$ has the form $\Gamma,\ l_c \vdash_s s$, where $\Gamma$ is the typing environment and $l_c$ is the secrecy context: true if the statement occurs within a

$$\frac{v \mapsto \tau \in \Gamma}{\Gamma \vdash_e v : \tau} \qquad \frac{\mathsf{id} \mapsto \tau_1, \ldots, \tau_n \to \tau \in \Gamma}{\Gamma \vdash_e \mathsf{id} : \tau_1, \ldots, \tau_n \to \tau} \qquad \frac{\exists r.\ \Gamma \vdash_e e : \mathsf{ref}\langle l_r \rangle\ r\ \mathsf{cell}\ \beta\langle l_v \rangle}{\Gamma \vdash_e \mathsf{deref}\ e : \beta\langle l_r \sqcup l_v \rangle}\ \text{LOAD}$$

$$\frac{\Gamma \vdash_e e : \mathsf{ref}\langle l \rangle\ r\ \mathsf{struct}\ \{\ldots, \mathsf{id} : \alpha, \ldots\}}{\Gamma \vdash_e e \rightsquigarrow \mathsf{id} : \mathsf{ref}\langle l \rangle\ r\ \alpha}\ \text{STRUCT} \qquad \frac{\Gamma \vdash_e e_a : \mathsf{ref}\langle l \rangle\ r\ \mathsf{array}\ n\ \alpha \qquad \Gamma \vdash_e e_i : \mathsf{idx}\langle n', l' \rangle \qquad n' \leq n}{\Gamma \vdash_e e_a[e_i] : \mathsf{ref}\langle l \sqcup l' \rangle\ r\ \alpha}\ \text{ARRAY}$$

$$\frac{\exists r.\ \Gamma \vdash_e e_a : \mathsf{ref}\ r\langle l_r \rangle\ \mathsf{cell}\ \beta\langle l_v \rangle \qquad \Gamma \vdash_e e_d : \beta\langle l_d \rangle \qquad l_d\ \sqcup\ l_r\ \sqcup\ l_c\ \sqsubseteq\ l_v}{\Gamma,\ l_c \vdash_s \mathsf{store}\ e_a\ e_d}\ \text{STORE}$$

$$\frac{\Gamma[v \mapsto \mathsf{ref}\langle \bot \rangle\ \mathsf{local}\ \alpha],\ \bot \vdash_s s}{\Gamma,\ \bot \vdash_s \mathsf{local}\ v : \alpha\ \mathsf{in}\ s}\ \text{LOCAL-BIND} \qquad \frac{\Gamma \vdash_e e : \tau \qquad \Gamma[v \mapsto \tau],\ l_c \vdash_s s}{\Gamma,\ l_c \vdash_s \mathsf{let}\ v = e\ \mathsf{in}\ s}\ \text{LET-BIND}$$

$$\frac{\Gamma \vdash_e e : \mathsf{bool}\langle l \rangle \qquad \Gamma,\ l_c \sqcup l \vdash_s s_1 \qquad \Gamma,\ l_c \sqcup l\ \vdash_s s_2}{\Gamma,\ l_c \vdash_s \mathsf{if}\ (e)\ \{s_1\}\ \mathsf{else}\ \{s_2\}}\ \text{COND} \qquad \frac{\Gamma \vdash_e e : \mathsf{bool}\langle \bot \rangle \qquad \Gamma, \bot \vdash_s s}{\Gamma,\ \bot \vdash_s \mathsf{while}(e)s}\ \text{WHILE}$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \ldots \Gamma \vdash_e e_n : \tau_n \qquad \Gamma \vdash_e \mathsf{id} : \tau_1, \ldots, \tau_n \to \tau \qquad \Gamma[v \mapsto \tau],\ \bot \vdash_s s}{\Gamma,\ \bot \vdash_s \mathtt{call}\ v = \mathsf{id}(e_1, \ldots, e_1)\ \mathsf{in}\ s}\ \text{CALL-BIND}$$

$$\frac{\Gamma,\ l_c \vdash_s s_1 \qquad \Gamma,\ l_c \vdash_s s_2}{\Gamma,\ l_c \vdash_s s_1; s_2}\ \text{SEQ} \qquad \frac{\Gamma \vdash_e e : \tau \qquad \neg\exists \alpha, l.\ \tau \neq \mathsf{ref}\langle l \rangle\ \mathsf{local}\ \alpha}{\Gamma,\ \bot \vdash_s \mathsf{ret}\ e}\ \text{RET}$$

$$\frac{\forall\ \mathsf{var}\ v : \alpha \in p.\ v \mapsto \mathsf{ref}\ \mathsf{global}\ \alpha \in \Gamma \qquad \Gamma[v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n],\ \bot \vdash_s s \qquad \text{each path in id ends in ret } e : \tau}{\Gamma \vdash_p \mathsf{proc}\ \mathsf{id}(v_1 : \tau_1, \ldots, v_n : \tau_n \to \tau)\{s\}}\ \text{PROC}$$

$$\frac{\forall\ \mathsf{proc}\ \mathsf{id}(v_1 : \tau_1, \ldots, v_n : \tau_n \to \tau)\{s\} \in p.\ \Gamma \vdash_p \mathsf{proc}\ \mathsf{id}(v_1 : \tau_1, \ldots, v_n : \tau_n \to \tau)\{s\})}{\vdash p}\ \text{PROGRAM}$$

*Figure 9.5*: Typing rules for EncLang. Typing environment $\Gamma ::= \emptyset \mid v \mapsto \tau,\ \Gamma$

$$
\begin{array}{lcll}
S_\phi[\![\text{local } v : \alpha \text{ in } s]\!] & = & S_\phi[\![s]\!] & \\
\\
S_\phi[\![\text{let } v = e \text{ in } s]\!] & = & E_\phi[\![e]\!] & \\
\phi' = (\phi_L, \phi_G, & & \text{mov rbx } \phi'_B[\text{v}] & \\
\quad \phi_B[v := \phi_\delta + 8], & & \text{mov [rbx] rax} & \\
\quad \phi_\delta + 8) & & S_{\phi'}[\![s]\!] & \\
\\
S_\phi[\![\text{store } e_a\ e_d]\!] & = & E_\phi[\![e_d]\!] & \\
 & & \text{mov rbx rax} & \{\,\text{rbx} \mapsto \Gamma(e_d)\,\} \\
 & & E_\phi[\![e_a]\!] & \\
 & & W[\![e_a]\!] & \\
\\
S_\phi[\![\texttt{call } v = \text{p}(e_1, \ldots, e_n) \text{ in } s]\!] & = & E_\phi[\![e_1]\!] & \\
\phi' = (\phi_L, \phi_G, \phi_B[v := \phi_\delta + 8] & & \text{push rax} & \\
\quad , \phi_\delta + 8) & & \ldots & \\
 & & E_\phi[\![e_n]\!] & \\
 & & \text{push rax} & \\
 & & \text{call p} & \\
 & & \text{push rax} & \\
 & & S_{\phi'}[\![s]\!] & \\
\\
S_\phi[\![\text{ret } e]\!] & = & E_\phi[\![e]\!] & \\
 & & \text{mov rsp rbp} & \\
 & & \text{pop rbp} & \\
 & & \text{pop rdi} & \\
 & & \text{pop rsi} & \\
\\
S_\phi[\![\text{if } (e)\ \{s_1\} \text{ else } \{s_2\}]\!] & = & E_\phi[\![e]\!] & \\
 & & \text{test rax rax} & \{\,\text{zf} \mapsto \Gamma(e = 0)\,\} \\
 & & \text{jz } l_{else} & \\
 & & l_{then}\text{: } S_\phi[\![s_1]\!] & \\
 & & \text{jmp } l_{end} & \\
 & & l_{else}\text{: } S_\phi[\![s_2]\!] & \\
 & & l_{end}\text{:} &
\end{array}
$$

$$
\begin{array}{lcll}
E_\phi[\![v]\!] & = & \text{mov rax } \phi_L[v] & \{\,\text{rax} \mapsto \Gamma(v)\,\} \\
\text{for } v \in \text{locals} & & & \\
\\
E_\phi[\![v]\!] & = & \text{mov rax } \phi_G[v] & \{\,\text{rax} \mapsto \Gamma(v)\,\} \\
\text{for } v \in \text{globals} & & & \\
\\
E_\phi[\![v]\!] & = & \text{mov rax } \phi_B[v] & \\
\text{for } v \in \text{bindings} & & \text{mov rax [rax]} & \{\,\text{rax} \mapsto \Gamma(v)\,\} \\
\\
E_\phi[\![\text{deref } e]\!] & = & E_\phi[\![e]\!] & \{\,\text{rax} \mapsto \Gamma(e)\,\} \\
 & & R[\![e]\!] & \{\,\text{rax} \mapsto \Gamma(\text{deref } e)\,\} \\
\\
E_\phi[\![e_1 \otimes e_2]\!] & = & E_\phi[\![e_2]\!] & \\
 & & \text{push rax} & \\
 & & E_\phi[\![e_1]\!] & \\
 & & \text{pop rdx} & \{\,\text{rdx} \mapsto \Gamma(e_2)\,\} \\
 & & \otimes \text{ rax rdx} & \{\,\text{rax} \mapsto \Gamma(e_1 \otimes e_2)\,\} \\
\\
E_\phi[\![e \rightsquigarrow \text{id}]\!] & = & E_\phi[\![e]\!] & \\
 & & \text{add rax off}(\Gamma(\text{e}), \text{id}) & \{\,\text{rax} \mapsto \Gamma(e \rightsquigarrow \text{id})\,\} \\
\\
E_\phi[\![e_a[e_i]]\!] & = & E_\phi[\![e_i]\!] & \\
 & & \text{imul rax sz}(\Gamma(\text{e}_\text{a})) & \{\,\text{rax} \mapsto \text{idx}\langle \text{sz}(\Gamma(e_a)) * n \rangle\,\} \\
 & & \text{push rax} & \\
 & & E_\phi[\![e_a]\!] & \\
 & & \text{pop rdx} & \{\,\text{rdx} \mapsto \text{idx}\langle \text{sz}(\Gamma(e_a)) * n \rangle\,\} \\
 & & \text{add rax rdx} & \{\,\text{rax} \mapsto \Gamma(e_a[e_i])\,\}
\end{array}
$$

$$
\begin{array}{lcl}
P_\phi[\![\text{proc p}(v_1, \ldots, v_n)\{s\}]\!] & = & \text{push rsi} \\
\phi' = (\phi_L, \phi_G, & & \text{push rdi} \\
\quad \phi_B[v_1 := \phi_\delta - 8, \ldots & & \text{push rbp} \\
\quad\quad v_n := \phi_\delta - 8 * n], & & \text{mov rbp rsp} \\
\quad 0) & & \langle\text{prologue}\rangle \\
 & & S_{\phi'}[\![s]\!]
\end{array}
$$

*Figure 9.6*: Compilation of EncLang to Typed Assembly Language. For any typing derivation $\Gamma \vdash_e e : \tau$, we use $\Gamma(e)$ to denote $\tau$.

secret-based conditional. A well-typed expression $e$ has a typing judgment $\Gamma \vdash_e e : \tau$, where $\tau$ is the inferred type. To check for valid information flows, we define a subtyping relation $\sqsubseteq$: $\bot$ is a subtype of $\top$, which creates a lattice with the join operator $\sqcup$: $l_1 \sqcup l_2$ is equal to $\bot$ if $l_1 = \bot \wedge l_2 = \bot$, and $\top$ otherwise.

The typing rules for LOAD and STORE mandate that deref and store recieve a reference to a primitive (cell) type, which is computed using a combination of array indexing and struct field accesses. In the case of store statements, the type checker checks that the secrecy type of reference ($l_r$) and the secrecy type of the written value ($l_d$) are subtypes of the referenced cell's secrecy type ($l_d$) — a secret reference can be thought of as a secret-based choice over a set of cells, and therefore must not be used to modify a public cell. The type system also checks that no store is made to a public cell in a secret context, which is a standard feature of type systems for non-interference. A load (deref) produces a secret value if it uses either a secret reference or a reference to a secret-valued cell. Both CALL-BIND and RET rules enforce that call and ret statements are performed in a public context — forcing calls (including malloc and free) to occur in a public context allows the PAO enforcement to be modular. The COND rule checks that both branches are well-typed in the security context determined by the branch condition. The LOOP rule forbids loops with a secret branching condition — secret (number of) loop iterations causes the (number of) page accesses to depend on a secret, and therefore hinders PAO enforcement. The ARRAY rule checks that the index expression of type idx⟨k⟩ does not cause out-of-bounds access on an array of size n by requiring $\mathsf{k} \leq \mathsf{n}$.

### 9.2.6 Compiling EncLang to Typed Assembly Language

To produce PAO-satisfying code, the compiler must control the placement of stack-allocated objects and instructions, using the algorithms from Section 9.2.1-Section 9.2.3. However, these algorithms operate at the level of read / write operations and machine instructions, which is produced after compilation. This circular dependency is broken by compiling the EncLang program in phases: 1) produce machine code with placeholders for the location of stack-allocated objects, 2) obliviate data accesses and optimize using MCMC sampling, which computes the location for stack-allocated objects, 3) obliviate code accesses, which computes the location for each instruction in the compiled program, and 4) assign placeholders from phase 1 using the locations computed in phase 2. The (nearly) entire implementation of the compiler's phase 1 is formalized in Figure 9.6; phases 2 and 3 are described in Section 9.2.1 and Section 9.2.3.

During phase 1, the compiler maintains a context $\phi = (\phi_L, \phi_G, \phi_B, \phi_\delta)$, which is read and modified during compilation to x86-64, as seen in Figure 9.6. Computed in phase 2, $\phi_L$ maps stack-allocated objects to locations in the current stack frame, specified as an offset relative to the frame pointer rbp — this is left as a placeholder in phase 1. $\phi_B$ tracks the location of bindings (produced by call and let statements), and is modified each time the compilation encounters such statement. $\phi_G$ maps globally-scoped objects to fixed, statically-computed addresses in the enclave address space.

The compiler is modular: each procedure p in the enclave is compiled independently by invoking $P_\phi[\![\mathsf{p}]\!]$. The procedure first pushes the callee-preserved registers on the stack, generates a code block called prologue (which we describe later), and invokes compilation on the procedure's body. A statement $s$ is compiled using $S_\phi[\![s]\!]$, which recursively compiles the constituent statements (using $S[\![]\!]$) and expressions (using $E[\![]\!]$), while making use of context $\phi$. For any expression $e$, $E_\phi[\![e]\!]$ stores the evaluation result in rax and is allowed to use rdx in any way. This process also produces $R[\![e_a]\!]$ and $W[\![e_a]\!]$ primitives, which is used in phase 2 to obliviate data accesses.

Recall that phase 2 produces $\phi_L$, a mapping from stack-allocated object to its location on the stack frame (potentially many pages), by using stochastic optimization to assign multiple objects onto a page. However, the compiler cannot compute the base address of the local stack frame (as the procedure can be called via an arbitrary call chain), and this hinders our ability to implement $\phi_L$. For this reason, the compiled program maintains two stacks: 1) a *bindings stack* used for storing bound variables (produced by a call or let) and intermediate results while evaluating expressions, and 2) a *locals stack* used for storing stack-allocated objects, which we will align at the page-boundary. The bindings stack bears resemblance to the stack used in a stack-based procedural language, and is accessed using the frame pointer rbp and stack pointer rsp. On the other hand, the locals stack is accessed relative to the register rsi, which we modify in the prologue to be the next page boundary, and it remains constant throughout the procedure — there is a caveat that if the procedure's stack space requirement can be met within the current page (e.g. it uses small objects), then we do not page-align rsi. This invariant on rsi enables the compiler to statically layout the objects to comply with the $\phi_L$ mapping and satisfy PAO.

Section 9.3 describes an independent verifier which certifies the compiled machine code. We follow the typed assembly language paradigm [85] of: 1) a strongly-typed source language, EncLang, 2) a type-preserving compiler, and 3) a strongly-typed assembly language (TAL from here on). As seen in the right-most column of Figure 9.6, each update to a register accompanies an annotation assigning its new type, derived using the type inference rules in Figure 9.5. The type annotations on registers, which are used for computing both references and values, simplifies information flow tracking and alias analysis of the otherwise unstructured, untyped machine code.

### 9.2.7 Supporting Heap Allocation and Procedure Calls

Since malloc enjoys complete control over the placement of objects within the heap, we obliviate heap accesses by performing $\hat{R}[\![.]\!]$ and $\hat{W}[\![.]\!]$ in the other branches of a secret conditional, using the technique described in Section 9.2.1. Procedure arguments are treated similarly, the compiler has no knowledge of their location. The $\hat{R}[\![.]\!]$ and $\hat{W}[\![.]\!]$ requires the size of the object to be statically known. Therefore, our malloc and free routines require the type of the requested object.

## 9.3 Verifying PAO

The verifier proves that for any pair of executions of the enclave binary (that only differ in secret values), the sequence of page accesses must be equivalent, where two accesses are equivalent if they target the same page and have same type (read / write / execute). The verifier independently analyzes each procedure because the compiler is modular and does not optimize globally. For each procedure, the verifier must:

**1. Enumerate secret-dependent paths** Enumerate all paths in the procedure such that each constituent path represents a unique evaluation of a secret-dependent conditional — number of paths is worst case exponential in the number of secret-dependent conditionals within the procedure. The compiler assists this step by providing secrecy types of CPU flags at all conditional jumps in the TAL program (see Figure 9.6). Furthermore, the lack of indirect jumps in the binary enables simple, yet precise control flow analysis.

**2. Identify sequence of memory accesses** : The verifier computes the sequence of memory accesses that the CPU performs on each path. This step is purely syntactic, and is implemented verbatim as the definition of PAO in Section 6.3.

**3. Identify target page(s) for each access** : The verifier computes the exact code page for each instruction, which is given verbatim in the enclave executable. Identifying the target pages for data accesses ordinarily necessitates an alias analysis which identifies the object(s) targeted by a reference, and recovering the mapping from objects to pages. We circumvent these analyses with a key insight: for any reference $e$, the verifier only needs to determine the statically-identifiable object $si(e)$ (defined in Section 9.2.1), for which the typing annotations are sufficient. In other words, the verifier can establish the equivalence of any two page accesses, which use references $e_1$ and $e_2$, knowing only $si(e_1)$ and $si(e_2)$. This is due to a combination of reasons: 1) our primitives $R[\![e]\!]$, etc. use only the knowledge of $si(e)$ to generate the memory accesses, 2) for any reference $e$, the semantics of EncLang ensures that $si(e)$ evaluates to the same object in all paths, and 3) the verifier is able to infer when different local objects ($si(e_1) \neq si(e_2)$) are mapped to the same page by leveraging the fact that all local objects are addressed relative to a fixed frame pointer.

Having performed these steps, the verifier asserts that the sequence of page accesses is equivalent in all the enumerated paths. Furthermore, to avoid trusting the compiler, the verifier also proves validity of the typing annotations (relative to the corresponding machine code) using a set of typing judgments.

## 9.4 Evaluation

To study the performance impact of automatic PAO enforcement, we implement an open-source toolchain consisting of a compiler from EncLang to TAL, and a verifier for certifying

| Benchmark | Code Size (no PAO) | Code Size (PAO) | Data Size (no PAO) | Data Size (PAO) |
|---|---|---|---|---|
| k-means | 1638 B | 1668 B | 1784 B | 1784 B |
| Decision Tree | 1058 B | 3082 B | 416 B | 416 B |
| SVM | 1368 B | 1408 B | 2008 B | 2008 B |
| CNN Classifier | 1637 B | 1667 B | 45312 B | 45312 B |
| IDCT (1 dim) | 4574 B | 10725 B | 592 B | 592 B |
| IDCT (2 dim) | 7424 B | 13575 B | 664 B | 664 B |
| AES Blk Cipher | 5173 B | 5203 B | 488 B | 488 B |

*Table 9.1*: Summary of results.

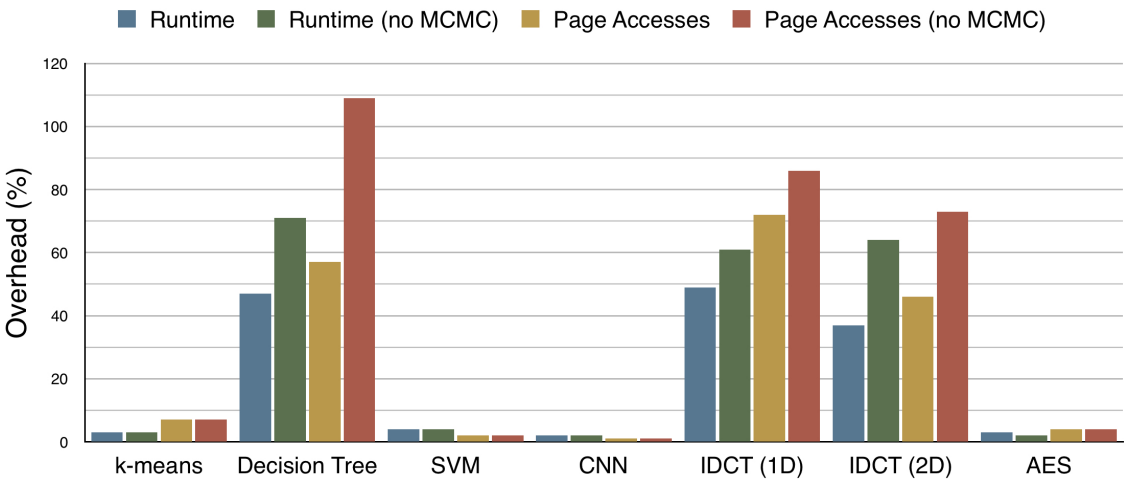



*Figure 9.7*: Overhead in Runtime and Page Accesses

PAO on the output TAL. Along with the TAL, the toolchain produces executables that we run natively on a 3.2 GHz 6th Generation Intel CPU (with SGX instruction set enabled and 96 MB available for enclave memory).

We evaluate this toolchain on several enclave programs that compute on sensitive data. We sample standard machine learning algorithms: k-means clustering, training of SVM classifier (based on a cache oblivious algorithm from [91]), decision tree evaluation (Figure 9.1), and a convolutional neural network (CNN). In the case of k-means clustering, the input points and the k=10 trained clusters must be kept secret. The training of SVM classifier must ensure that the learned weights are kept secret. The CNN (trained offline) must ensure that the image to be classified must be kept secret. We also experiment with the inverse discrete cosine transform (IDCT) routine (both one and two dimensions) from a JPEG decoder, whose page fault patterns were exploited by Xu et al. [125] to infer edges in the secret input images. To compare with [110], we also evaluate on the AES block cipher (we only evaluate encryption).

The compiler takes roughly 2-3 seconds to compile each of these benchmarks, most of which is spent in the MCMC optimization. The verifier uses the typing annotations generated by the compiler in order to produce the proof, and this takes under 1 second for all of these benchmarks. Table 9.1 compares vanilla compilation with PAO-enforcing compilation, with respect to memory consumption for code and data pages. Across all the benchmarks, we observe an average of 81 % increase in code size, and 0 % increase in memory requirements because the compiler successfully rearranged local objects without introducing any padding space. That being said, the runtime overhead is significantly lower than code size overhead (81 %) because the added x64 instructions are distributed across multiple control branches.

Figure 9.7 reports the performance overheads for runtime and number of page accesses, using standard datasets (e.g. UCI Repository [3]) for machine learning programs and $10^6$ invocations with randomly generated inputs for IDCT and AES programs. The overhead denotes the increase in these metrics when enforcing PAO, and is reported after averaging over 10 runs on a large dataset, or a million invocations . We observe a non-negative overhead (%) because our algorithm instruments dummy accesses, which lead to additional data and code accesses (to fetch the added instructions). In general, we find that programs with more conditional branches (in a secret context) incur higher code size overheads because the EncLang compiler places dummy code and data accesses to determinize the page access sequence across all branches, which leads to additional x86 instructions. Specifically, IDCT incurs higher runtime overheads because the `libjpeg` implementation skips the complex computation when the input image satisfies a condition [125], and such optimization violates PAO because the attacker observes different page accesses based on secret input. Similarly, the decision tree classification incurs a high overhead because the oblivious implementation must perform a fixed number of tree traversals for all inputs. On the other hand, we find that k-means, SVM, CNN classification, and AES incur low overheads as the computation is data-intensive and primarily occurs outside of secret conditionals. Specifically, CNN and AES show negligible overhead as all array accesses use public indices, and loops use constant bounds; our compiler infers that they have no secret-dependent code or data accesses.

## 9.5 Related Work

Shinde at al. [110] develop an instrumentation scheme (at the LLVM IR level) for enforcing page fault obliviousness — computational security definition as opposed to non-interference — of programs written in a subset of C/C++. Their approach, termed deterministic multiplexing, copies all code and data blocks at the same level of the execution "tree" to a temporary page, and dynamically selects the appropriate code and data based on the current program path. Although [110] addresses the same side channel attack as this paper, there are several important differences. First, we protect against a stronger adversary that can observe all page accesses, whereas the attacker in [110] only observes changes in page accesses, i.e., the attacker cannot count accesses within a single page. A realistic attack scenario can invalidate a page table entry at any time (e.g. OS getting interrupted by a

device). Therefore, the defense in [110] potentially leaks a bit of information on each page access, which is problematic for long-running enclaves. Second, we develop a binary verifier to certify PAO. On the contrary, [110] makes unspecified assumptions such as memory safety — a control flow exploit can bypass their instrumentation, and worse, spill secrets directly to untrusted memory. Furthermore, [110] incurs a large software TCB comprising the C compiler, LLVM assembler, and their instrumentation scheme. Our software TCB contains just the TAL verifier. Finally, [110] obliviates by transforming the compiled LLVM program. While this allows them to target mainstream languages, their scheme incurs 705x average runtime overhead — they reduce this overhead using developer annotations, which may compromise soundness. We show that by carefully designing the semantics and compiler of EncLang, we can optimize the PAO enforcement to incur an average 49% overhead across various benchmarks.

Ohrimenko et al. [91] manually develop machine learning algorithms that guarantee data-obliviousness at a cache-line granularity; their ideas inspired our primitives for oblivious dummy accesses. Cache side-channel defenses are complementary to our work because pages and cache sets are addressed by disjoint bits in a virtual address.

Oblivious RAM (ORAM) [54] protects against side channel leaks via the program's memory access patterns. Liu et al. [72] formalize memory trace obliviousness, and develop a compiler for producing memory trace oblivious programs by partitioning code and data across multiple ORAM banks for efficiency; in a follow-up work, Liu et al. [73] develop ObliVM, a tool to compile high-level source programs to an oblivious representation that leverages ORAMs to efficiently perform dynamic memory accesses (in lieu of performing a linear scan). GhostRider [70] presents a co-designed compiler and hardware ORAM for memory trace oblivious execution. Although these techniques provide stronger guarantees than our work, they require a novel hardware platform with ORAM support, whereas we target commodity SGX machines. An interesting question arises whether an ORAM controller can be implemented within an SGX enclave in our threat model. As [110] explains, ORAM constructions require a private stash for shuffling data blocks, whereas our attacker can observe accesses to all pages in memory. Furthermore, to adapt the ORAM algorithm (e.g. Path ORAM) in our compiler, we need to "compose" the ORAM logic with the enclave program, effectively evaluating the ORAM logic on each instruction. Our compiler can be seen as a static scheme for achieving obliviousness, and it uses the type system and program behavior to optimize the performance overheads.

## 9.6 Summary

This chapter develops tools for enforcing PAO (Definition 4) automatically. The toolchain comprises 1) a compiler for EncLang that automatically enforces PAO, 2) a stochastic optimization to reduce the runtime overhead, and 3) a verifier to certify the compiled machine code. The toolchain provably guarantees PAO while achieving a tiny trusted computing base, which only includes the enclave platform and the verifier's implementation.

# Chapter 10

# Conclusion and Future Work

This chapter summarizes the contributions of this thesis and proposes several directions for future research.

## 10.1 Closing Statement

Security-critical applications that compute on sensitive data constantly face threats due to vulnerabilities in the application's code, and privileged software layers including the OS, Hypervisor, etc. With the development of trusted hardware primitives by several processor vendors, applications can be designed with separate untrusted and trusted components, which are called enclaves — in this paradigm, enclaves are the only trusted software components, and hence, must not contain any vulnerabilities that can be exploited to leak sensitive data to the adversary. The developer must exercise extreme care to ensure that all of the enclave's interaction with the untrusted platform are safe (e.g., by encrypting outputs and persistent storage) — this includes any side channels exposed by the specific hardware platform (e.g. page-level access patterns in SGX). This thesis makes several novel contributions towards techniques for building and verifying enclave programs with confidentiality guarantees, while accounting for the effects of a privileged software-level adversary.

Since we are developing techniques for writing safe enclaves, the first part of this thesis develops a formal framework for reasoning about an enclave's semantics i.e. its set of behaviors in the presence of a privileged adversary. Chapter 3 develops a formal semantics of enclave execution, which precisely defines the enclave's state, inputs and outputs, and the model of execution. In order to ensure that enclaves behave according to their semantics on real platforms, and in the presence of a privileged attacker, we formally specify the behavior of the hardware primitives in Chapter 4. Furthermore, to generalize our analysis beyond a specific enclave platform, we develop an abstract model (called Trusted Abstract Platform, or TAP in short) and show that popular platforms such as Intel SGX and MIT Sanctum are instantiations of the TAP. We also formalize the operations and observations of the privileged attacker. Finally, in Chapter 5, we prove that the TAP provides secure remote execution to any enclave. That is, any enclave that is launched on a TAP enjoys integrity, confidentiality,

and secure measurement properties, which ensures that enclaves behave according to the expected semantics that we defined in Chapter 3.

The second part of the thesis develops novel techniques for designing and verifying enclave programs with provable confidentiality properties, even in the presence of a privileged adversary. Since an enclave communicates with remote entities via the untrusted platform, any output produced by an enclave is visible to the adversary. To address potential leaks of sensitive data via such outputs, we develop a machine code verifier in Chapter 7, called Moat, that accepts the enclave's binary as input, statically analyzes all potential executions, and either produces a proof of confidentiality or flags a potential leak. Although Moat is sound, it does not scale beyond tiny enclave programs due to its approach of precisely tracking the flow of secrets in memory, at the level of machine code. We address these scalability limitations in Chapter 8, where we impose a design constraint (termed information release confinement) that all interaction between a potentially vulnerable enclave and the untrusted platform take place via a trusted runtime library — the library automatically enforces cryptographic protections, such as encryption of all outbound messages and file I/O. With this design constraint, we simplify the verification to only prove that the developer-provided enclave module does not write directly to non-enclave memory, nor violate a form of control flow integrity — we empirically demonstrate that this verification is significantly simpler than the one performed by Moat, and hence, also much more scalable. Note that both Moat and /CONFIDENTIAL provide confidentiality against an attacker that observes an enclave's output (i.e., writes to non-enclave memory). Finally, we note that some hardware platforms expose certain side channel leaks e.g. cache timing, memory access patterns — these side channels are exposed on SGX but not Sanctum. To address leaks via the side channel of page-level memory access pattern, in Chapter 9, we develop a compiler that automatically produces safe machine code (i.e. page access sequence is independent of the enclave's secrets), and a verifier that further proves that the compiler's output code is indeed safe, thus removing the compiler from the trusted computing base. The contributions of Chapter 9 allow us to develop enclave programs that protect sensitive data in the presence of a privileged software adversary that also observes page-level access patterns.

## 10.2 Future Work

We conclude with a discussion of future research directions influenced by the work presented in this thesis.

**Applications** An important subject of future research is the design of cloud-based systems, such as databases, data analytics systems, and web servers, that leverage trusted hardware to provide strong security (e.g. confidentiality, integrity, freshness) guarantees while maintaining the level of performance available in off-the-shelf, unprotected systems. While there has been some research in building database engines and Map-Reduce frameworks [107] that use enclave platforms to perform security-critical functions, the performance impact has been

significant.

**Application Partitioning** Throughout this thesis, we presented several applications and benchmarks with enclave components, which we manually developed by partitioning the application into untrusted and trusted components, where the trusted components were run within enclaves. However, for most real-world applications, such partitioning is a significant manual undertaking, and any programming error (that does not move all of the sensitive computation to the enclave) can be exploited by the adversary. Important directions for future research include tools and techniques to enable the developer to (semi) automatically partition applications into trusted and untrusted components, while guaranteeing confidentiality properties and minimal performance overheads. Undoubtedly, these tools would require hints or annotations from the developer (e.g. which state variables hold secret data), in which case, further research is needed on how to enable the developer to express such hints.

**Hardware Verification** In this thesis, we formally specify the semantics of trusted hardware primitives, and simply assume that the hardware implementation satisfies this specification. For instance, we formalize the property of secure remote execution on an enclave platform, which decomposes into sub-properties (such as confidentiality, integrity, and secure measurement) of the hardware. Verifying that the low-level hardware implementation satisfies these properties, which are actually formalized as hyper-properties, requires further research in modeling and verification.

# Bibliography


[1] Available at http://www.cryptopp.com/.

[2] Codebases millions of lines of code. http://www.informationisbeautiful.net/visualizations/million-lines-of-code/.

[3] UCI Machine Learning Repository. https://archive.ics.uci.edu/ml.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[5] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1 – 70, 1999.

[6] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[7] T. Alves and D. Felton. TrustZone: Integrated Hardware and Software Security. *Information Quarterly*, 3(4):18–24, 2004.

[8] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, volume 13, 2013.

[9] ARM Security Technology - Building a Secure System using TrustZone Technology. ARM Technical White Paper.

[10] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[11] K. Asanović and D. A. Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[12] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. *Theor. Comput. Sci.*, 402(2-3):82–101, July 2008.

[13] M. Balliu, M. Dam, and R. Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1080–1091, New York, NY, USA, 2014. ACM.

[14] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.

[15] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05*, pages 82–87, 2005.

[16] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.

[17] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[18] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[19] G. Barthe and L. P. Nieto. Secure information flow for a concurrent language with scheduling. In *Journal of Computer Security*, pages 647–689. IOS Press, 2007.

[20] S. Bauer, P. Cuoq, and J. Regehr. Deniable backdoors using compiler bugs. *International Journal of PoC||GTFO*, 0x08:7–9, June 2015.

[21] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[22] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corp., 1975.

[23] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, 1977.

[24] J. Black, J. Rogaway, and T. Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *SAC*, 2002.

[25] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop*, pages 82–96, Cape Breton, Canada, June 2001.

[26] B. Blanchet. A computationally sound automatic prover for cryptographic protocols. In *Workshop on the link between formal and computational models*, Paris, France, June 2005.

[27] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[28] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. *CoRR*, abs/1702.07521, 2017.

[29] Brian Donohue. Linux Vulnerable to Apple Cert Bug (Sort of). https://www.kaspersky.com/blog/linux-vulnerable-to-apple-cert-bug-sort-of/4063/.

[30] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59:115–131, 1988.

[31] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *CAV*, 2011.

[32] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, 2011.

[33] S. Checkoway and H. Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. volume 41, pages 253–264, New York, NY, USA, Mar. 2013. ACM.

[34] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008.

[35] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 85–90, New York, NY, USA, 2009. ACM.

[36] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, Sept. 2010.

[37] V. Costan and S. Devadas. Intel SGX explained. *Cryptology ePrint Archive, Report 2016/086*, 2016. http://eprint.iacr.org/2016/086.

[38] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium*, pages 857–874, Austin, TX, 2016. USENIX Association.

[39] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for c and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 648–664, New York, NY, USA, 2016. ACM.

[40] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *ASPLOS*, 2014.

[41] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[42] R. DeLine and K. R. M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[43] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[44] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[45] I. S. Diatchki and M. P. Jones. Strongly typed memory areas programming systems-level data structures in a functional language. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 72–83, New York, NY, USA, 2006. ACM.

[46] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *Transactions on Architecture and Code Optimization (TACO)*, 2012.

[47] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488, 2014.

[48] T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury. Guilt free ivory. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 189–200, New York, NY, USA, 2015. ACM.

[49] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: software guards for system address spaces. In *OSDI*, 2006.

[50] A. Fontaine, P. Chifflier, and T. Coudray. Picon : Control flow integrity on llvm ir. In *SSTIC*, 2015.

[51] C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings 35th Symposium on Principles of Programming Languages*. G. Smith, 2008.

[52] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 465–482, Berlin, Heidelberg, 2010. Springer-Verlag.

[53] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[54] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[55] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, GA, 2016. USENIX Association.

[56] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: end-to-end security via automated full-system verification. In *OSDI*, 2014.

[57] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 261–269. ACM, 2010.

[58] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP*, volume 13, 2013.

[59] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[60] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure applications on an untrusted operating system. In *ASPLOS*, 2013.

[61] Intel Software Guard Extensions Programming Reference. Available at https://software.intel.com/sites/default/files/329298-001.pdf, 2014.

[62] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[63] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. https://github.com/Cr4sh/ThinkPwn.git.

[64] R. Joshi and R. Leino. A semantic approach to secure information flow. volume 37 (2000), pages 113–138. Elsevier, January 2000.

[65] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an OS kernel. In *SOSP*, 2009.

[66] B. Krebs. Fear not: You, too, are a cybercrime victim! https://krebsonsecurity.com/tag/equifax/.

[67] B. Lampson. A note on the confinment problem. *Communications of the ACM*, 16(10), 1976.

[68] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.

[69] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, 2000.

[70] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Not.*, 50(4):87–101, Mar. 2015.

[71] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[72] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 51–65, Washington, DC, USA, 2013. IEEE Computer Society.

[73] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 359–376, Washington, DC, USA, 2015. IEEE Computer Society.

[74] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, Mar 2016.

[75] F. Liu and R. B. Lee. Random Fill Cache Architecture. In *Microarchitecture (MICRO)*. IEEE, 2014.

[76] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.

[77] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, 2015.

[78] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Usenix Security*, 2008.

[79] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. *HASP*, 13:10, 2013.

[80] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[81] J. Mclean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1:37–58, 1992.

[82] A. Moghimi, G. Irazoqui, and T. Eisenbarth. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *CoRR*, abs/1703.06986, 2017.

[83] R. Morisset, P. Pawan, and F. Z. Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, 2013.

[84] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *PLDI*, 2012.

[85] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[86] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[87] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[88] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.

[89] B. Niu and G. Tan. Modular control flow integrity. In *PLDI*, 2014.

[90] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, 2013.

[91] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, Austin, TX, Aug. 2016. USENIX Association.

[92] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical State Continuity for Protected Modules. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 379–394, Washington, DC, USA, 2011. IEEE Computer Society.

[93] R. Pass, E. Shi, and F. Tramèr. Formal Abstractions for Attested Execution Secure Processors. *IACR Cryptology ePrint Archive*, 2016:1027, 2016.

[94] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, 2015.

[95] M. Patrignani and D. Clarke. Fully abstract trace semantics for low-level isolation mechanisms. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1562–1569, 2014.

[96] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.

[97] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[98] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995*, pages 114–127, 1995.

[99] Ryan Paul. Linux kernel in 2011: 15 million total lines of code and Microsoft is a top contributor. https://arstechnica.com/information-technology/2012/04/linux-kernel-in-2011-15-million-total-lines-of-code.

[100] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[101] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *In Proc. International Symp. on Software Security*, pages 174–191. Springer-Verlag, 2004.

[102] J. Salowey, A. Choudhury, and D. McGrew. Aes-gcm cipher suites for tls. IETF: TLS Working Group, 2008.

[103] J. H. Saltzer and M. D. Schroeder. Formal verification of a realistic compiler. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[104] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 67–80. ACM, 2014.

[105] Satisfiability Modulo Theories Library (SMT-LIB). Available at http://smt-lib.org/.

[106] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. *SIGPLAN Not.*, 48(4):305–316, Mar. 2013.

[107] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *36th IEEE Symposium on Security and Privacy*. IEEE âĂŞ Institute of Electrical and Electronics Engineers, May 2015.

[108] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *Usenix Security*, 2010.

[109] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 317–328, 2016.

[110] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 317–328, New York, NY, USA, 2016. ACM.

[111] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 665–681, New York, NY, USA, 2016. ACM.

[112] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *CCS*, 2015.

[113] R. Sinha, S. Rajamani, and S. A. Seshia. A compiler and verifier for page access oblivious computation. Technical Report UCB/EECS-2017-124, EECS Department, University of California, Berkeley, Jul 2017.

[114] T. Skolem. Logico-combinatorial investigations in the satisfiability or provability of mathematical propositions: a simplified proof of a theorem by L. Löwenheim and generalizations of the theorem. *From Frege to Gödel. A Source Book in Mathematical Logic, 1879-1931*, pages 252–263, 1967.

[115] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *CAV*, 2011.

[116] P. Subramanyan, R. Sinha, I. A. Lebedev, S. Devadas, and S. A. Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2435–2450, 2017.

[117] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[118] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis Symposium (SAS '05)*, LNCS 3672, pages 352–367, 2005.

[119] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.

[120] E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[121] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan. 1996.

[122] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[123] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović. The risc-v instruction set manual volume ii: Privileged architecture version 1.9.1. Technical Report UCB/EECS-2016-161, EECS Department, University of California, Berkeley, Nov 2016.

[124] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[125] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.

[126] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, pages 15–15, Berkeley, CA, USA, 2012. USENIX Association.

[127] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE*, 2008.

[128] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[129] Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732, 2014.

[130] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *S&P*, 2009.

[131] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22d Symposium on Operating Systems Principles*, SOSP '09, pages 291–304, New York, NY, USA, 2009. ACM.

[132] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS*, 2011.

[133] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. Armor: Fully verified software fault isolation. In *EMSOFT*, 2011.

[134] A. Zylva. Windows 10 device guard and credential guard demystified. https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/.

# Appendix A

# TAP Model

In this appendix chapter, we present our model of the TAP in full detail, which was described earlier in Chapter 4. We include the reference implementation (in BoogiePL [14]) of all TAP operations: launch, enter, resume, exit, pause, and destroy.

```
1  procedure launch(
2    /* eid.            */ eid           : tap_enclave_id_t,
3    /* VA to PA. mapping */ addr_valid   : addr_valid_t,
4    /* VA to PA mapping */ addr_map     : addr_map_t,
5    /* excl vaddr      */ excl_vaddr    : excl_vaddr_t,
6    /* excl paddr      */ excl_paddr    : excl_map_t,
7    /* entrypoint.     */ entrypoint    : vaddr_t
8  )
9     returns (status : enclave_op_result_t)
10 {
11    var i, k           : int;
12    var mappings_alias_v : bool;
13    var paddr          : wap_addr_t;
14    var va             : vaddr_t;
15    var cache_conflict : bool;
16
17    // ensure cpu mode is valid.
18    if (cpu_enclave_id != tap_null_enc_id) {
19       status := enclave_op_invalid_arg; return;
20    }
21    // ensure eid is valid.
22    if (!valid_enclave_id(eid) || tap_enclave_metadata_valid[eid]) {
23       status := enclave_op_invalid_arg; return;
24    }
25    // the entrypoint must be mapped and exclusive.
26    if (!tap_addr_perm_x(addr_valid[entrypoint]) || !excl_paddr[addr_map[entrypoint]] ||
          !excl_vaddr[entrypoint]) {
27       status := enclave_op_invalid_arg; return;
28    }
29    // Ensure none of the paddr's are already exclusive.
30    paddr := k0_wap_addr_t;
31    ...
```

*Figure A.1*: Reference implementation of launch: only first half listed here, with second half presented in Figure A.2

```
    ...
    while (LT_wapa(paddr, kmax_wap_addr_t)) {
        if (excl_paddr[paddr]) {
            if (cpu_owner_map[paddr] != tap_null_enc_id) {
                status := enclave_op_invalid_arg; return;
            }
        }
        if (cpu_owner_map[paddr] == eid) {
            status := enclave_op_invalid_arg; return;
        }
        paddr := PLUS_wapa(paddr, k1_wap_addr_t);
    }

    if ((excl_paddr[paddr] && cpu_owner_map[paddr] != tap_null_enc_id) ||
         (cpu_owner_map[paddr] == eid)) {
        status := enclave_op_invalid_arg; return;
    }
    // check if the private addresses alias with anything else (paddr).
    call mappings_alias_v := do_mappings_alias_v(excl_vaddr, addr_map);
    if (mappings_alias_v) {
        status := enclave_op_invalid_arg; return;
    }

    // check if the private virt address map to a shared phys addr
    va := k0_vaddr_t;
    while (LT_va(va, kmax_vaddr_t)) {
        if (excl_vaddr[va]) {
            if(!excl_paddr[addr_map[va]] || !tap_addr_perm_v(addr_valid[va])) {
                status := enclave_op_invalid_arg;
                return;
            }
        }
        va := PLUS_va(va, k1_vaddr_t);
    }
    if (excl_vaddr[va] && (!excl_paddr[addr_map[va]] || !tap_addr_perm_v(addr_valid[va]))) {
        status := enclave_op_invalid_arg; return;
    }
    // Set the CPU owner map.
    paddr := k0_wap_addr_t;
    while (LT_wapa(paddr, kmax_wap_addr_t)) {
        if (excl_paddr[paddr]) { cpu_owner_map[paddr] := eid; }
        paddr := PLUS_wapa(paddr, k1_wap_addr_t);
    }
    if (excl_paddr[paddr]) { cpu_owner_map[paddr] := eid; }

    // regs are zeroed out.
    call cache_conflict := does_enclave_conflict(eid);

    tap_enclave_metadata_valid[eid]      := true;
    tap_enclave_metadata_addr_map[eid]   := addr_map;
    tap_enclave_metadata_addr_valid[eid] := addr_valid;
    tap_enclave_metadata_addr_excl[eid]  := excl_vaddr;
    tap_enclave_metadata_entrypoint[eid] := entrypoint;
    tap_enclave_metadata_pc[eid]         := entrypoint;
    tap_enclave_metadata_regs[eid]       := kzero_regs_t;
    tap_enclave_metadata_paused[eid]     := false;
    tap_enclave_metadata_cache_conflict[eid] := cache_conflict;

    status := enclave_op_success;
}
```

*Figure A.2*: Reference implementation of launch: second half listed here, with first half presented in Figure A.1

```
1  procedure enter(eid: tap_enclave_id_t) returns (status : enclave_op_result_t)
2  {
3      if (!valid_enclave_id(eid) ||
4          !tap_enclave_metadata_valid[eid] ||
5          cpu_enclave_id != tap_null_enc_id)
6      {
7          status := enclave_op_invalid_arg;
8          return;
9      }
10
11     status                  := enclave_op_success;
12     // save context.
13     untrusted_regs          := cpu_regs;
14     untrusted_addr_valid    := cpu_addr_valid;
15     untrusted_addr_map      := cpu_addr_map;
16     untrusted_pc            := cpu_pc;
17     // restore enclave context.
18     cpu_enclave_id          := eid;
19     cpu_addr_valid          := tap_enclave_metadata_addr_valid[eid];
20     cpu_addr_map            := tap_enclave_metadata_addr_map[eid];
21     cpu_pc                  := tap_enclave_metadata_entrypoint[eid];
22 }
```

*Figure A.3*: Reference implementation of enter, used to transfer control to an enclave

```
1  procedure resume(eid: tap_enclave_id_t)
2      returns (status : enclave_op_result_t)
3
4  {
5      if (!valid_enclave_id(eid)          ||
6          !tap_enclave_metadata_valid[eid] ||
7          !tap_enclave_metadata_paused[eid] ||
8          cpu_enclave_id != tap_null_enc_id)
9      {
10         status := enclave_op_invalid_arg;
11         return;
12     }
13
14     // save context.
15     untrusted_regs              := cpu_regs;
16     untrusted_addr_valid        := cpu_addr_valid;
17     untrusted_addr_map          := cpu_addr_map;
18     untrusted_pc                := cpu_pc;
19     // restore enclave context.
20     cpu_enclave_id              := eid;
21     cpu_addr_valid              := tap_enclave_metadata_addr_valid[eid];
22     cpu_addr_map                := tap_enclave_metadata_addr_map[eid];
23     cpu_pc                      := tap_enclave_metadata_pc[eid];
24     cpu_regs                    := tap_enclave_metadata_regs[eid];
25     status                      := enclave_op_success;
26 }
```

*Figure A.4*: Reference implementation of resume, used to resume execution within an enclave following an asynchronous interrupt

```
implementation exit()
    returns (status : enclave_op_result_t)
{
    var eid : tap_enclave_id_t;

    // no enclave id is null.
    if (!valid_enclave_id(cpu_enclave_id) || !tap_enclave_metadata_valid[cpu_enclave_id]) {
        status := enclave_op_failed; return;
    }

    status := enclave_op_success;

    eid                               := cpu_enclave_id;
    tap_enclave_metadata_addr_valid[eid] := cpu_addr_valid;
    tap_enclave_metadata_addr_map[eid] := cpu_addr_map;
    tap_enclave_metadata_pc[eid]      := tap_enclave_metadata_entrypoint[eid];
    tap_enclave_metadata_paused[eid] := false;

    cpu_enclave_id := tap_null_enc_id;
    cpu_regs      := untrusted_regs;
    cpu_addr_valid := untrusted_addr_valid;
    cpu_addr_map  := untrusted_addr_map;
    cpu_pc        := untrusted_pc;
    status        := enclave_op_success;
}
```

*Figure A.5*: Reference implementation of exit, used by enclave to transfer control back to the untrusted calling code

```
procedure pause()
    returns (status : enclave_op_result_t)
{
    var eid : tap_enclave_id_t;

    // no enclave id is null.
    if (!valid_enclave_id(cpu_enclave_id) || !tap_enclave_metadata_valid[cpu_enclave_id]) {
        status := enclave_op_failed; return;
    }

    status := enclave_op_success;

    eid                               := cpu_enclave_id;
    tap_enclave_metadata_regs[eid]    := cpu_regs;
    tap_enclave_metadata_addr_valid[eid] := cpu_addr_valid;
    tap_enclave_metadata_addr_map[eid] := cpu_addr_map;
    tap_enclave_metadata_pc[eid]      := cpu_pc;
    tap_enclave_metadata_paused[eid] := true;

    cpu_enclave_id := tap_null_enc_id;
    cpu_regs      := untrusted_regs;
    cpu_addr_valid := untrusted_addr_valid;
    cpu_addr_map  := untrusted_addr_map;
    cpu_pc        := untrusted_pc;
    status        := enclave_op_success;
}
```

*Figure A.6*: Reference implementation of pause, used by the untrusted OS to interrupt the enclave. This models an asynchronous interrupt e.g. timer or device interrupt.

```
1 procedure destroy(eid: tap_enclave_id_t)
2     returns (status: enclave_op_result_t)
3
4 {
5     var pa : wap_addr_t;
6     // no enclave id is null.
7     if (!valid_enclave_id(eid) || !tap_enclave_metadata_valid[eid] || cpu_enclave_id != tap_null_enc_id) {
8         status := enclave_op_invalid_arg;
9         return;
10    }
11
12    // we have to clear out the enclaves registers and memory.
13    pa := k0_wap_addr_t;
14    while (LT_wapa(pa, kmax_wap_addr_t)) {
15        if (cpu_owner_map[pa] == eid) {
16            cpu_owner_map[pa] := tap_blocked_enc_id;
17        }
18        pa := PLUS_wapa(pa, k1_wap_addr_t);
19    }
20    if (cpu_owner_map[kmax_wap_addr_t] == eid) {
21        cpu_owner_map[kmax_wap_addr_t] := tap_blocked_enc_id;
22    }
23
24    // and now we mark the enclave invalid.
25    tap_enclave_metadata_valid[eid] := false;
26    tap_enclave_metadata_regs[eid] := kzero_regs_t;
27    tap_enclave_metadata_pc[eid] := k0_vaddr_t;
28
29    status := enclave_op_success;
30 }
```

*Figure A.7*: Reference implementation of destroy, used to tear down an enclave

# Appendix B

# Model of Intel SGX

In this appendix chapter, we present our model of Intel SGX in full detail, which was described earlier in Chapter 4.

```
//******************** Types ********************
type sgx_api_result_t = bv1;
type page_table_map_t = [vaddr_t] wap_addr_t;
type page_table_valid_t = [vaddr_t] bool;
//processor id
type core_id_t = int;
type core_state_t;
type gpregs_t; //registers, but left abstract
type lr_register_t;
type page_t = bv2;
type epcm_entry_t;
type key_t = int;
type sgx_measurement_t = int;
type hashtext_t a; //unary type constructor
type ciphertext_t a;
type mactext_t a;
type attributes_t;
type targetinfo_t;
type report_t;
type report_maced_t;
type keyname_t;
type keyrequest_t;
type einittoken_t;
type sigstruct_t;
type sigstruct_signature_t = ciphertext_t (hashtext_t sigstruct_t);
type sigstruct_signed_t;
type secinfo_t;
type pcmd_t;
type pageinfo_t;
type secs_t;
type tcs_t;

//******************** States ********************
var page_table_map : page_table_map_t;
var page_table_valid : page_table_valid_t;
var curr_core : core_id_t;
var gpregs : [core_id_t] gpregs_t;
var core_state : [core_id_t] core_state_t;
var mem_secs : [wap_addr_t] secs_t;
var mem_tcs : [wap_addr_t] tcs_t;
var mem_reg : [wap_addr_t] word_t;
var epcm : [wap_addr_t] epcm_entry_t;
var arbitrary_write_count: int;

//******************** Constants ********************
const CSR_INTELPUBKEYHASH : hashtext_t key_t;
const EPC_LOW : wap_addr_t; axiom EPC_LOW =4096bv22; //arbitrary value
const EPC_HIGH : wap_addr_t; axiom EPC_HIGH =45056bv22; //arbitrary value
const PAGE_SIZE: vaddr_t; axiom PAGE_SIZE =12bv32;

const dummy_signing_key : key_t;
const sgx_api_invalid_value : sgx_api_result_t; axiom sgx_api_invalid_value =0bv1;
const sgx_api_success : sgx_api_result_t; axiom sgx_api_success =1bv1;
const abort_page : wap_addr_t; axiom abort_page =0bv22;
const dummy_lsrr : lr_register_t;
```

```
const dummy_gpregs : gpregs_t;
const pt_secs : page_t; axiom pt_secs =0bv2;
const pt_tcs : page_t; axiom pt_tcs =1bv2;
const pt_reg : page_t; axiom pt_reg =2bv2;

//******************** Functions *********************

function {: inline} pageof_va(va: vaddr_t) : vaddr_t { va[32: 12] ++ 0bv12 }
function {: inline} pageof_pa(pa: wap_addr_t) : wap_addr_t { pa[22: 12] ++ 0bv12 }

//linear range register: (lbase, lsize)
function Lr_register(lbase: vaddr_t, lsize: vaddr_t) : lr_register_t;
function Lr_register_lbase (lr : lr_register_t) : vaddr_t;
function Lr_register_lsize (lr : lr_register_t) : vaddr_t;
axiom (∀ lbase: vaddr_t, lsize: vaddr_t ●{Lr_register(lbase,lsize)}
      Lr_register_lbase(Lr_register(lbase, lsize)) =lbase);
axiom (∀ lbase: vaddr_t, lsize: vaddr_t ●{Lr_register(lbase,lsize)}
      Lr_register_lsize(Lr_register(lbase, lsize)) =lsize);
axiom (∀ lr: lr_register_t ●{Lr_register_lbase(lr)} {Lr_register_lsize(lr)}
      Lr_register(Lr_register_lbase(lr), Lr_register_lsize(lr)) =lr);

function in_register_range (vaddr_t, lr_register_t) : bool;
axiom (∀ la : vaddr_t, lr : lr_register_t ●{in_register_range(la,lr)}
      in_register_range(la,lr) ⟺
      (LE_va(Lr_register_lbase(lr), la) ∧
       LT_va(la, PLUS_va(Lr_register_lbase(lr), Lr_register_lsize(lr)))));

//Enclave related memory: all physical memory is partitioned into EPC memory and non-EPC memory
function is_epc_address (wap_addr_t) : bool;
axiom (∀ i: wap_addr_t ●{is_epc_address(i)}
     is_epc_address(i) ⟺(LE_wapa(EPC_LOW, i) ∧LT_wapa(i, EPC_HIGH)));

//******************** Processor *********************
//processor state is type-cast to data when storing to memory
function gpregs_to_word(gpregs_t) : word_t;
function word_to_gpregs(word_t) : gpregs_t;
axiom (∀ w: word_t ●{word_to_gpregs(w)}
   gpregs_to_word(word_to_gpregs(w)) =w);
axiom (∀ x: gpregs_t ●{gpregs_to_word(x)}
   word_to_gpregs(gpregs_to_word(x)) =x);

function Core_state(cr_enclave_mode: bool,
                cr_tcs_pa: wap_addr_t,
                cr_active_secs: wap_addr_t,
                cr_elrange: lr_register_t,
                ssa_pa: wap_addr_t)
                : core_state_t;
function Core_state_cr_enclave_mode (cores : core_state_t) : bool;
function Core_state_cr_tcs_pa (cores : core_state_t) : wap_addr_t;
function Core_state_cr_active_secs (cores : core_state_t) : wap_addr_t;
function Core_state_cr_elrange (cores : core_state_t) : lr_register_t;
function Core_state_ssa_pa (cores : core_state_t) : wap_addr_t;
axiom (∀ cr_enclave_mode: bool,
            cr_tcs_pa: wap_addr_t,
            cr_active_secs: wap_addr_t,
            cr_elrange: lr_register_t,
            ssa_pa: wap_addr_t ●
            {Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)}
      Core_state_cr_enclave_mode(Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)) =cr_enclave_mode);
axiom (∀ cr_enclave_mode: bool,
            cr_tcs_pa: wap_addr_t,
            cr_active_secs: wap_addr_t,
            cr_elrange: lr_register_t,
            ssa_pa: wap_addr_t ●
            {Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)}
      Core_state_cr_tcs_pa(Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)) =cr_tcs_pa);
axiom (∀ cr_enclave_mode: bool,
            cr_tcs_pa: wap_addr_t,
            cr_active_secs: wap_addr_t,
            cr_elrange: lr_register_t,
            ssa_pa: wap_addr_t ●
            {Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)}
      Core_state_cr_active_secs(Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)) =cr_active_secs);
axiom (∀ cr_enclave_mode: bool,
            cr_tcs_pa: wap_addr_t,
            cr_active_secs: wap_addr_t,
            cr_elrange: lr_register_t,
            ssa_pa: wap_addr_t ●
            {Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)}
      Core_state_cr_elrange(Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)) =cr_elrange);
axiom (∀ cr_enclave_mode: bool,
            cr_tcs_pa: wap_addr_t,
            cr_active_secs: wap_addr_t,
            cr_elrange: lr_register_t,
            ssa_pa: wap_addr_t ●
            {Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)}
```

```
      Core_state_ssa_pa(Core_state(cr_enclave_mode, cr_tcs_pa, cr_active_secs, cr_elrange, ssa_pa)) =ssa_pa);

axiom (∀ cores: core_state_t ●
             {Core_state_cr_enclave_mode(cores)}
             {Core_state_cr_tcs_pa(cores)}
             {Core_state_cr_active_secs(cores)}
             {Core_state_cr_elrange(cores)}
             {Core_state_ssa_pa(cores)}
       Core_state( Core_state_cr_enclave_mode(cores),
                Core_state_cr_tcs_pa(cores),
                Core_state_cr_active_secs(cores),
                Core_state_cr_elrange(cores),
                Core_state_ssa_pa(cores) ) =cores);


//******************** SGX structs *********************

function Attributes(einittokenkey: bool): attributes_t;
function Attributes_einittokenkey(attributes: attributes_t): bool;
axiom (∀ einittokenkey: bool ●
  {Attributes(einittokenkey)}
  Attributes_einittokenkey(Attributes(einittokenkey)) =einittokenkey);
axiom (∀ attributes: attributes_t ●
  {Attributes_einittokenkey(attributes)}
  Attributes(Attributes_einittokenkey(attributes)) =attributes);

function Targetinfo(attributes: attributes_t, measurement: sgx_measurement_t): targetinfo_t;
function Targetinfo_attributes(targetinfo: targetinfo_t): attributes_t;
function Targetinfo_measurement(targetinfo: targetinfo_t): sgx_measurement_t;
axiom (∀ attributes: attributes_t, measurement: sgx_measurement_t ●
  {Targetinfo(attributes, measurement)}
  Targetinfo_attributes(Targetinfo(attributes, measurement)) =attributes);
axiom (∀ attributes: attributes_t, measurement: sgx_measurement_t ●
  {Targetinfo(attributes, measurement)}
  Targetinfo_measurement(Targetinfo(attributes, measurement)) =measurement);
axiom (∀ targetinfo: targetinfo_t ●
  {Targetinfo_attributes(targetinfo)}
  {Targetinfo_measurement(targetinfo)}
  Targetinfo(Targetinfo_attributes(targetinfo), Targetinfo_measurement(targetinfo)) =targetinfo);

function Report(isvprodid: int, isvsvn: int, attributes: attributes_t,
              reportdata: word_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t) : report_t;
function Report_isvprodid(report: report_t): int;
function Report_isvsvn(report: report_t): int;
function Report_attributes(report: report_t): attributes_t;
function Report_reportdata(report: report_t): word_t;
function Report_mrenclave(report: report_t): sgx_measurement_t;
function Report_mrsigner(report: report_t): hashtext_t key_t;
axiom (∀ isvprodid: int, isvsvn: int, attributes: attributes_t,
            reportdata: word_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t ●
  {Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)}
  Report_isvprodid(Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)) =isvprodid);
axiom (∀ isvprodid: int, isvsvn: int, attributes: attributes_t,
            reportdata: word_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t ●
  {Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)}
  Report_isvsvn(Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)) =isvsvn);
axiom (∀ isvprodid: int, isvsvn: int, attributes: attributes_t,
            reportdata: word_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t ●
  {Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)}
  Report_attributes(Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)) =attributes);
axiom (∀ isvprodid: int, isvsvn: int, attributes: attributes_t,
            reportdata: word_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t ●
  {Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)}
  Report_reportdata(Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)) =reportdata);
axiom (∀ isvprodid: int, isvsvn: int, attributes: attributes_t,
            reportdata: word_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t ●
  {Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)}
  Report_mrenclave(Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)) =mrenclave);
axiom (∀ isvprodid: int, isvsvn: int, attributes: attributes_t,
            reportdata: word_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t ●
  {Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)}
  Report_mrsigner(Report(isvprodid, isvsvn, attributes, reportdata, mrenclave, mrsigner)) =mrsigner);
axiom (∀ report: report_t ●
  {Report_isvprodid(report)}
  {Report_isvsvn(report)}
  {Report_attributes(report)}
  {Report_reportdata(report)}
  {Report_mrenclave(report)}
  {Report_mrsigner(report)}
  Report(Report_isvprodid(report),
         Report_isvsvn(report),
         Report_attributes(report),
         Report_reportdata(report),
         Report_mrenclave(report),
         Report_mrsigner(report)) =report);
```

```
function Report_maced(report: report_t, mac: mactext_t report_t): report_maced_t;
function Report_maced_report(report_maced: report_maced_t): report_t;
function Report_maced_mac(report_maced: report_maced_t): mactext_t report_t;
axiom (∀ report: report_t, mac: mactext_t report_t •
  {Report_maced(report, mac)}
  Report_maced_report(Report_maced(report, mac)) =report);
axiom (∀ report: report_t, mac: mactext_t report_t •
  {Report_maced(report, mac)}
  Report_maced_mac(Report_maced(report, mac)) =mac);
axiom (∀ report_maced: report_maced_t •
  {Report_maced_report(report_maced)}
  {Report_maced_mac(report_maced)}
  Report_maced(Report_maced_report(report_maced),
               Report_maced_mac(report_maced)) =report_maced);

const unique launch_key : keyname_t;
const unique provision_key : keyname_t;
const unique provision_seal_key : keyname_t;
const unique report_key : keyname_t;
const unique seal_key : keyname_t;

function Keyrequest(keyname: keyname_t, isvsvn: int,
                    keypolicy_mrenclave: bool, keypolicy_mrsigner: bool): keyrequest_t;
function Keyrequest_keyname(keyrequest: keyrequest_t) : keyname_t;
function Keyrequest_isvsvn(keyrequest: keyrequest_t) : int;
function Keyrequest_keypolicy_mrenclave(keyrequest: keyrequest_t) : bool;
function Keyrequest_keypolicy_mrsigner(keyrequest: keyrequest_t) : bool;
axiom (∀ keyname: keyname_t, isvsvn: int,
            keypolicy_mrenclave: bool, keypolicy_mrsigner: bool •
  {Keyrequest(keyname, isvsvn, keypolicy_mrenclave, keypolicy_mrsigner)}
  Keyrequest_keyname(Keyrequest(keyname, isvsvn, keypolicy_mrenclave, keypolicy_mrsigner)) =keyname);
axiom (∀ keyname: keyname_t, isvsvn: int,
            keypolicy_mrenclave: bool, keypolicy_mrsigner: bool •
  {Keyrequest(keyname, isvsvn, keypolicy_mrenclave, keypolicy_mrsigner)}
  Keyrequest_isvsvn(Keyrequest(keyname, isvsvn, keypolicy_mrenclave, keypolicy_mrsigner)) =isvsvn);
axiom (∀ keyname: keyname_t, isvsvn: int,
            keypolicy_mrenclave: bool, keypolicy_mrsigner: bool •
  {Keyrequest(keyname, isvsvn, keypolicy_mrenclave, keypolicy_mrsigner)}
  Keyrequest_keypolicy_mrenclave(Keyrequest(keyname, isvsvn, keypolicy_mrenclave, keypolicy_mrsigner)) =keypolicy_mrenclave);
axiom (∀ keyname: keyname_t, isvsvn: int,
            keypolicy_mrenclave: bool, keypolicy_mrsigner: bool •
  {Keyrequest(keyname, isvsvn, keypolicy_mrenclave, keypolicy_mrsigner)}
  Keyrequest_keypolicy_mrsigner(Keyrequest(keyname, isvsvn, keypolicy_mrenclave, keypolicy_mrsigner)) =keypolicy_mrsigner);
axiom (∀ keyrequest: keyrequest_t •
  {Keyrequest_keyname(keyrequest)}
  {Keyrequest_isvsvn(keyrequest)}
  {Keyrequest_keypolicy_mrenclave(keyrequest)}
  {Keyrequest_keypolicy_mrsigner(keyrequest)}
  Keyrequest(Keyrequest_keyname(keyrequest),
             Keyrequest_isvsvn(keyrequest),
             Keyrequest_keypolicy_mrenclave(keyrequest),
             Keyrequest_keypolicy_mrsigner(keyrequest)) =keyrequest);

function Einittoken(valid: bool, attributes: attributes_t,
                    mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t) : einittoken_t;
function Einittoken_valid(einittoken: einittoken_t): bool;
function Einittoken_attributes(einittoken: einittoken_t): attributes_t;
function Einittoken_mrenclave(einittoken: einittoken_t): sgx_measurement_t;
function Einittoken_mrsigner(einittoken: einittoken_t): hashtext_t key_t;
axiom (∀ valid: bool, attributes: attributes_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t •
  {Einittoken(valid, attributes, mrenclave, mrsigner)}
  Einittoken_valid(Einittoken(valid, attributes, mrenclave, mrsigner)) =valid);
axiom (∀ valid: bool, attributes: attributes_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t •
  {Einittoken(valid, attributes, mrenclave, mrsigner)}
  Einittoken_attributes(Einittoken(valid, attributes, mrenclave, mrsigner)) =attributes);
axiom (∀ valid: bool, attributes: attributes_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t •
  {Einittoken(valid, attributes, mrenclave, mrsigner)}
  Einittoken_mrenclave(Einittoken(valid, attributes, mrenclave, mrsigner)) =mrenclave);
axiom (∀ valid: bool, attributes: attributes_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t •
  {Einittoken(valid, attributes, mrenclave, mrsigner)}
  Einittoken_mrsigner(Einittoken(valid, attributes, mrenclave, mrsigner)) =mrsigner);
axiom (∀ einittoken: einittoken_t •
  {Einittoken_valid(einittoken)}
  {Einittoken_attributes(einittoken)}
  {Einittoken_mrenclave(einittoken)}
  {Einittoken_mrsigner(einittoken)}
  Einittoken(Einittoken_valid(einittoken),
             Einittoken_attributes(einittoken),
             Einittoken_mrenclave(einittoken),
             Einittoken_mrsigner(einittoken)) =einittoken);

function Sigstruct(modulus: key_t, enclavehash: sgx_measurement_t,
                   attributes: attributes_t, isvprodid: int, isvsvn: int) : sigstruct_t;
function Sigstruct_modulus(sigstruct_t) : key_t;
function Sigstruct_enclavehash(sigstruct_t): sgx_measurement_t;
```

```
function Sigstruct_attributes(sigstruct_t): attributes_t;
function Sigstruct_isvprodid(sigstruct_t): int;
function Sigstruct_isvsvn(sigstruct_t): int;
axiom (∀ modulus: key_t, enclavehash: sgx_measurement_t,
            attributes: attributes_t, isvprodid: int, isvsvn: int ●
  {Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)}
  Sigstruct_modulus(Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)) =modulus);
axiom (∀ modulus: key_t, enclavehash: sgx_measurement_t,
            attributes: attributes_t, isvprodid: int, isvsvn: int ●
  {Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)}
  Sigstruct_enclavehash(Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)) =enclavehash);
axiom (∀ modulus: key_t, enclavehash: sgx_measurement_t,
            attributes: attributes_t, isvprodid: int, isvsvn: int ●
  {Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)}
  Sigstruct_attributes(Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)) =attributes);
axiom (∀ modulus: key_t, enclavehash: sgx_measurement_t,
            attributes: attributes_t, isvprodid: int, isvsvn: int ●
  {Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)}
  Sigstruct_isvprodid(Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)) =isvprodid);
axiom (∀ modulus: key_t, enclavehash: sgx_measurement_t,
            attributes: attributes_t, isvprodid: int, isvsvn: int ●
  {Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)}
  Sigstruct_isvsvn(Sigstruct(modulus, enclavehash, attributes, isvprodid, isvsvn)) =isvsvn);
axiom (∀ sigstruct: sigstruct_t ●
  {Sigstruct_modulus(sigstruct)}
  {Sigstruct_enclavehash(sigstruct)}
  {Sigstruct_attributes(sigstruct)}
  {Sigstruct_isvprodid(sigstruct)}
  {Sigstruct_isvsvn(sigstruct)}
  Sigstruct(Sigstruct_modulus(sigstruct),
           Sigstruct_enclavehash(sigstruct),
           Sigstruct_attributes(sigstruct),
           Sigstruct_isvprodid(sigstruct),
           Sigstruct_isvsvn(sigstruct)) =sigstruct);

function Sigstruct_signed(signature: sigstruct_signature_t, sigstruct: sigstruct_t) : sigstruct_signed_t;
function Sigstruct_signed_signature(sigstruct_signed_t): sigstruct_signature_t;
function Sigstruct_signed_sigstruct(sigstruct_signed_t): sigstruct_t;
axiom (∀ signature: sigstruct_signature_t, sigstruct: sigstruct_t ●
  {Sigstruct_signed(signature, sigstruct)}
  Sigstruct_signed_signature(Sigstruct_signed(signature, sigstruct)) =signature);
axiom (∀ signature: sigstruct_signature_t, sigstruct: sigstruct_t ●
  {Sigstruct_signed(signature, sigstruct)}
  Sigstruct_signed_sigstruct(Sigstruct_signed(signature, sigstruct)) =sigstruct);
axiom (∀ sigstruct_signed: sigstruct_signed_t ●
  {Sigstruct_signed_signature(sigstruct_signed)}
  {Sigstruct_signed_sigstruct(sigstruct_signed)}
  Sigstruct_signed(Sigstruct_signed_signature(sigstruct_signed),
                Sigstruct_signed_sigstruct(sigstruct_signed)) =sigstruct_signed);

function Secinfo(flags_r: bool, flags_w: bool, flags_x: bool, flags_pt: page_t): secinfo_t;
function Secinfo_flags_r(secinfo: secinfo_t): bool;
function Secinfo_flags_w(secinfo: secinfo_t): bool;
function Secinfo_flags_x(secinfo: secinfo_t): bool;
function Secinfo_flags_pt(secinfo: secinfo_t) : page_t;
axiom (∀ flags_r: bool, flags_w: bool, flags_x: bool, flags_pt: page_t ●
     {Secinfo(flags_r, flags_w, flags_x, flags_pt)}
     Secinfo_flags_r(Secinfo(flags_r, flags_w, flags_x, flags_pt)) =flags_r);
axiom (∀ flags_r: bool, flags_w: bool, flags_x: bool, flags_pt: page_t ●
     {Secinfo(flags_r, flags_w, flags_x, flags_pt)}
     Secinfo_flags_w(Secinfo(flags_r, flags_w, flags_x, flags_pt)) =flags_w);
axiom (∀ flags_r: bool, flags_w: bool, flags_x: bool, flags_pt: page_t ●
     {Secinfo(flags_r, flags_w, flags_x, flags_pt)}
     Secinfo_flags_x(Secinfo(flags_r, flags_w, flags_x, flags_pt)) =flags_x);
axiom (∀ flags_r: bool, flags_w: bool, flags_x: bool, flags_pt: page_t ●
     {Secinfo(flags_r, flags_w, flags_x, flags_pt)}
     Secinfo_flags_pt(Secinfo(flags_r, flags_w, flags_x, flags_pt)) =flags_pt);
axiom (∀ secinfo: secinfo_t ●
      {Secinfo_flags_r(secinfo)}
      {Secinfo_flags_w(secinfo)}
      {Secinfo_flags_x(secinfo)}
      {Secinfo_flags_pt(secinfo)}
      Secinfo(Secinfo_flags_r(secinfo),
             Secinfo_flags_w(secinfo),
             Secinfo_flags_x(secinfo),
             Secinfo_flags_pt(secinfo)) =secinfo);

function Pageinfo(linaddr: vaddr_t, srcpge: vaddr_t,
               secinfo: secinfo_t, pcmd: pcmd_t, secs: wap_addr_t): pageinfo_t;
function Pageinfo_linaddr(pageinfo: pageinfo_t) : vaddr_t;
function Pageinfo_srcpge(pageinfo: pageinfo_t) : vaddr_t;
function Pageinfo_secinfo(pageinfo: pageinfo_t) : secinfo_t;
function Pageinfo_pcmd(pageinfo: pageinfo_t) : pcmd_t;
function Pageinfo_secs(pageinfo: pageinfo_t) : wap_addr_t;
axiom (∀ linaddr: vaddr_t, srcpge: vaddr_t,
            secinfo: secinfo_t, pcmd: pcmd_t, secs: wap_addr_t ●
```

```
      {Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)}
      Pageinfo_linaddr(Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)) =linaddr);
axiom (∀ linaddr: vaddr_t, srcpge: vaddr_t,
            secinfo: secinfo_t, pcmd: pcmd_t, secs: wap_addr_t •
      {Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)}
      Pageinfo_srcpge(Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)) =srcpge);
axiom (∀ linaddr: vaddr_t, srcpge: vaddr_t,
            secinfo: secinfo_t, pcmd: pcmd_t, secs: wap_addr_t •
      {Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)}
      Pageinfo_secinfo(Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)) =secinfo);
axiom (∀ linaddr: vaddr_t, srcpge: vaddr_t,
            secinfo: secinfo_t, pcmd: pcmd_t, secs: wap_addr_t •
      {Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)}
      Pageinfo_pcmd(Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)) =pcmd);
axiom (∀ linaddr: vaddr_t, srcpge: vaddr_t,
            secinfo: secinfo_t, pcmd: pcmd_t, secs: wap_addr_t •
      {Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)}
      Pageinfo_secs(Pageinfo(linaddr, srcpge, secinfo, pcmd, secs)) =secs);
axiom (∀ pageinfo: pageinfo_t •
      {Pageinfo_linaddr(pageinfo)}
      {Pageinfo_srcpge(pageinfo)}
      {Pageinfo_secinfo(pageinfo)}
      {Pageinfo_pcmd(pageinfo)}
      {Pageinfo_secs(pageinfo)}
      Pageinfo(Pageinfo_linaddr(pageinfo),
               Pageinfo_srcpge(pageinfo),
               Pageinfo_secinfo(pageinfo),
               Pageinfo_pcmd(pageinfo),
               Pageinfo_secs(pageinfo)) =pageinfo);

function Secs(baseaddr: vaddr_t, size: vaddr_t, initialized: bool,
            mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t,
            isvprodid: int, isvsvn: int,
            attributes: attributes_t) : secs_t;
function Secs_baseaddr (secs : secs_t) : vaddr_t;
function Secs_size (secs: secs_t) : vaddr_t;
function Secs_initialized (secs: secs_t) : bool;
function Secs_mrenclave (secs: secs_t) : sgx_measurement_t;
function Secs_mrsigner (secs: secs_t) : hashtext_t key_t;
function Secs_isvprodid (secs: secs_t) : int;
function Secs_isvsvn (secs: secs_t) : int;
function Secs_attributes (secs: secs_t) : attributes_t;
axiom (∀ baseaddr: vaddr_t, size: vaddr_t, initialized: bool,
            mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t,
            isvprodid: int, isvsvn: int, attributes: attributes_t •
      {Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)}
      Secs_baseaddr(Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)) =baseaddr);
axiom (∀ baseaddr: vaddr_t, size: vaddr_t, initialized: bool,
            mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t,
            isvprodid: int, isvsvn: int, attributes: attributes_t •
      {Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)}
      Secs_size(Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)) =size);
axiom (∀ baseaddr: vaddr_t, size: vaddr_t, initialized: bool,
            mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t,
            isvprodid: int, isvsvn: int, attributes: attributes_t •
      {Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)}
      Secs_initialized(Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)) =initialized);
axiom (∀ baseaddr: vaddr_t, size: vaddr_t, initialized: bool,
            mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t,
            isvprodid: int, isvsvn: int, attributes: attributes_t •
      {Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)}
      Secs_mrenclave(Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)) =mrenclave);
axiom (∀ baseaddr: vaddr_t, size: vaddr_t, initialized: bool,
            mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t,
            isvprodid: int, isvsvn: int, attributes: attributes_t •
      {Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)}
      Secs_mrsigner(Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)) =mrsigner);
axiom (∀ baseaddr: vaddr_t, size: vaddr_t, initialized: bool,
            mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t,
            isvprodid: int, isvsvn: int, attributes: attributes_t •
      {Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)}
      Secs_isvprodid(Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)) =isvprodid);
axiom (∀ baseaddr: vaddr_t, size: vaddr_t, initialized: bool,
            mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t,
            isvprodid: int, isvsvn: int, attributes: attributes_t •
      {Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)}
      Secs_isvsvn(Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)) =isvsvn);
axiom (∀ baseaddr: vaddr_t, size: vaddr_t, initialized: bool,
            mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t,
            isvprodid: int, isvsvn: int, attributes: attributes_t •
      {Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)}
      Secs_attributes(Secs(baseaddr, size, initialized, mrenclave, mrsigner, isvprodid, isvsvn, attributes)) =attributes);
axiom (∀ secs: secs_t •
      {Secs_baseaddr(secs)}
      {Secs_size(secs)}
      {Secs_initialized(secs)}
```

```
      {Secs_mrenclave(secs)}
      {Secs_mrsigner(secs)}
      {Secs_isvprodid(secs)}
      {Secs_isvsvn(secs)}
      {Secs_attributes(secs)}
      Secs(Secs_baseaddr(secs), Secs_size(secs), Secs_initialized(secs),
          Secs_mrenclave(secs), Secs_mrsigner(secs),
          Secs_isvprodid(secs), Secs_isvsvn(secs), Secs_attributes(secs)) =secs);

function Tcs(active: bool, interrupted: bool, ossa: vaddr_t, nssa: vaddr_t, cssa: vaddr_t) : tcs_t;
function Tcs_active (tcs : tcs_t) : bool;
function Tcs_interrupted (tcs : tcs_t) : bool;
function Tcs_ossa (tcs : tcs_t) : vaddr_t;
function Tcs_nssa (tcs : tcs_t) : vaddr_t;
function Tcs_cssa (tcs : tcs_t) : vaddr_t;
axiom (∀ active: bool, interrupted: bool, ossa: vaddr_t, nssa: vaddr_t, cssa: vaddr_t •
      {Tcs(active, interrupted, ossa, nssa, cssa)}
      Tcs_active(Tcs(active, interrupted, ossa, nssa, cssa)) =active);
axiom (∀ active: bool, interrupted: bool, ossa: vaddr_t, nssa: vaddr_t, cssa: vaddr_t •
      {Tcs(active, interrupted, ossa, nssa, cssa)}
      Tcs_interrupted(Tcs(active, interrupted, ossa, nssa, cssa)) =interrupted);
axiom (∀ active: bool, interrupted: bool, ossa: vaddr_t, nssa: vaddr_t, cssa: vaddr_t •
      {Tcs(active, interrupted, ossa, nssa, cssa)}
      Tcs_ossa(Tcs(active, interrupted, ossa, nssa, cssa)) =ossa);
axiom (∀ active: bool, interrupted: bool, ossa: vaddr_t, nssa: vaddr_t, cssa: vaddr_t •
      {Tcs(active, interrupted, ossa, nssa, cssa)}
      Tcs_nssa(Tcs(active, interrupted, ossa, nssa, cssa)) =nssa);
axiom (∀ active: bool, interrupted: bool, ossa: vaddr_t, nssa: vaddr_t, cssa: vaddr_t •
      {Tcs(active, interrupted, ossa, nssa, cssa)}
      Tcs_cssa(Tcs(active, interrupted, ossa, nssa, cssa)) =cssa);
axiom (∀ tcs: tcs_t •
      {Tcs_active(tcs)}
      {Tcs_interrupted(tcs)}
      {Tcs_ossa(tcs)}
      {Tcs_nssa(tcs)}
      {Tcs_cssa(tcs)}
      Tcs(Tcs_active(tcs), Tcs_interrupted(tcs), Tcs_ossa(tcs), Tcs_nssa(tcs), Tcs_cssa(tcs)) =tcs);

//this is meant to be used for writes made by hardware within an SGX instruction. They don't need access permission checks.
function arbitrary_secs_val(int): secs_t;
function arbitrary_tcs_val(int): tcs_t;
function arbitrary_reg_val(int): word_t;

procedure {: inline 1} unchecked_write_secs(pa: wap_addr_t, val: secs_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs;
{
  mem_secs[pa] :=val;
  mem_reg[pa] :=arbitrary_reg_val(arbitrary_write_count);
  mem_tcs[pa] :=arbitrary_tcs_val(arbitrary_write_count);
  arbitrary_write_count :=arbitrary_write_count + 1;
}

procedure {: inline 1} unchecked_write_tcs(pa: wap_addr_t, val: tcs_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs;
{
  mem_tcs[pa] :=val;
  mem_reg[pa] :=arbitrary_reg_val(arbitrary_write_count);
  mem_secs[pa] :=arbitrary_secs_val(arbitrary_write_count);
  arbitrary_write_count :=arbitrary_write_count + 1;
}

procedure {: inline 1} unchecked_write_reg(pa: wap_addr_t, val: word_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs;
{
  mem_reg[pa] :=val;
  mem_secs[pa] :=arbitrary_secs_val(arbitrary_write_count);
  mem_tcs[pa] :=arbitrary_tcs_val(arbitrary_write_count);
  arbitrary_write_count :=arbitrary_write_count + 1;
}

function Epcm(valid: bool, r: bool, w: bool, x: bool, pt: page_t, enclavesecs: wap_addr_t, enclaveaddress: vaddr_t) : epcm_entry_t;
function Epcm_valid(epcm_entry : epcm_entry_t) : bool;
function Epcm_R(epcm_entry : epcm_entry_t): bool;
function Epcm_W(epcm_entry : epcm_entry_t): bool;
function Epcm_X(epcm_entry : epcm_entry_t): bool;
function Epcm_pt (epcm_entry : epcm_entry_t) : page_t;
function Epcm_enclavesecs (epcm_entry : epcm_entry_t) : wap_addr_t;
function Epcm_enclaveaddress (epcm_entry : epcm_entry_t) : vaddr_t;
axiom (∀ valid: bool, r: bool, w: bool, x: bool, pt: page_t, enclavesecs: wap_addr_t, enclaveaddress: vaddr_t •
      {Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)}
      Epcm_valid(Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)) =valid);
axiom (∀ valid: bool, r: bool, w: bool, x: bool, pt: page_t, enclavesecs: wap_addr_t, enclaveaddress: vaddr_t •
      {Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)}
      Epcm_R(Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)) =r);
axiom (∀ valid: bool, r: bool, w: bool, x: bool, pt: page_t, enclavesecs: wap_addr_t, enclaveaddress: vaddr_t •
      {Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)}
```

```
      Epcm_W(Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)) =w);
axiom (∀ valid: bool, r: bool, w: bool, x: bool, pt: page_t, enclavesecs: wap_addr_t, enclaveaddress: vaddr_t ●
      {Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)}
      Epcm_X(Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)) =x);
axiom (∀ valid: bool, r: bool, w: bool, x: bool, pt: page_t, enclavesecs: wap_addr_t, enclaveaddress: vaddr_t ●
      {Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)}
      Epcm_pt(Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)) =pt);
axiom (∀ valid: bool, r: bool, w: bool, x: bool, pt: page_t, enclavesecs: wap_addr_t, enclaveaddress: vaddr_t ●
      {Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)}
      Epcm_enclavesecs(Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)) =enclavesecs);
axiom (∀ valid: bool, r: bool, w: bool, x: bool, pt: page_t, enclavesecs: wap_addr_t, enclaveaddress: vaddr_t ●
      {Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)}
      Epcm_enclaveaddress(Epcm(valid, r, w, x, pt, enclavesecs, enclaveaddress)) =enclaveaddress);
axiom (∀ epcm_entry: epcm_entry_t ●
      {Epcm_valid(epcm_entry)}
      {Epcm_R(epcm_entry)}
      {Epcm_W(epcm_entry)}
      {Epcm_X(epcm_entry)}
      {Epcm_pt(epcm_entry)}
      {Epcm_enclavesecs(epcm_entry)}
      {Epcm_enclaveaddress(epcm_entry)}
      Epcm(Epcm_valid(epcm_entry),
           Epcm_R(epcm_entry),
           Epcm_W(epcm_entry),
           Epcm_X(epcm_entry),
           Epcm_pt(epcm_entry),
           Epcm_enclavesecs(epcm_entry),
           Epcm_enclaveaddress(epcm_entry)) =
      epcm_entry);

const dummy_epcm : epcm_entry_t;
axiom dummy_epcm =Epcm(false, false, false, false, pt_reg, abort_page, 0bv32);


procedure {: inline 1} is_accessible(core: core_id_t, la: vaddr_t) returns (result: bool)
{
  var pa : wap_addr_t; //pagetable[la]
  var ea : bool; //is this access to enclave memory?
  var mapped_la : bool; //does pagetable map this to an address ≠abort_page

  ea :=Core_state_cr_enclave_mode(core_state[core]) ∧
       in_register_range(la, Core_state_cr_elrange(core_state[core]));
  pa :=page_table_map[la];
  mapped_la :=page_table_valid[la];
  result :=mapped_la ∧
           (ea ⟹((Epcm_valid(epcm[pageof_pa(pa)]) ∧is_epc_address(pa)) ∧
                    Epcm_pt(epcm[pageof_pa(pa)]) =pt_reg ∧
                    Epcm_enclavesecs(epcm[pageof_pa(pa)]) =Core_state_cr_active_secs(core_state[core]) ∧
                    Epcm_enclaveaddress(epcm[pageof_pa(pa)]) =pageof_va(la)));
}

procedure {: inline 1} translate(la: vaddr_t) returns (result: wap_addr_t)
{
  var ea : bool;
  var pa : wap_addr_t;
  var accessible : bool;

  call accessible :=is_accessible(curr_core, la);

  ea :=Core_state_cr_enclave_mode(core_state[curr_core]) ∧
       in_register_range(la, Core_state_cr_elrange(core_state[curr_core]));

  pa :=page_table_map[la];

  if (¬page_table_valid[la] ∨¬accessible ∨(¬ea ∧is_epc_address(pa))) {
    result :=abort_page;
  } else {
    result :=pa;
  }
}

//******************** Helper predicates ********************
//Is cpu represented by cores running an enclave thread whose secs is pa?
function thread_in_enclave(core_state_t, wap_addr_t) : bool;
axiom (∀ cores: core_state_t, pa: wap_addr_t ●
  {thread_in_enclave(cores, pa)}
  thread_in_enclave(cores,pa) ⟺
  (Core_state_cr_enclave_mode(cores) ∧(Core_state_cr_active_secs(cores) =pa)));

function no_threads_in_enclave([core_id_t] core_state_t, wap_addr_t) : bool;
axiom (∀ core_state : [core_id_t] core_state_t, pa: wap_addr_t ●
  {no_threads_in_enclave(core_state, pa)}
  no_threads_in_enclave(core_state, pa) ⟺
  (∀ core: core_id_t ●¬thread_in_enclave(core_state[core], pa)));

function page_in_enclave(epcm_entry_t, wap_addr_t, wap_addr_t) : bool;
```

```
axiom (∀ epcm_entry : epcm_entry_t, pa: wap_addr_t, ps: wap_addr_t ●
  {page_in_enclave(epcm_entry, pa, ps)}
  page_in_enclave(epcm_entry, pa, ps) ⟺
  (is_epc_address(pa) ∧
   Epcm_valid(epcm_entry) ∧
   Epcm_enclavesecs(epcm_entry) =ps ∧
   pa ≠ps));

function no_pages_in_enclave([wap_addr_t] epcm_entry_t, wap_addr_t) : bool;
axiom (∀ epcm: [wap_addr_t] epcm_entry_t, ps: wap_addr_t ●
  {no_pages_in_enclave(epcm, ps)}
  no_pages_in_enclave(epcm, ps) ⟺
  (∀ pa: wap_addr_t ●¬page_in_enclave(epcm[pageof_pa(pa)], pa, ps)));

function cssa_addr(secs_t, tcs_t) : vaddr_t;
axiom (∀ secs: secs_t, tcs: tcs_t ●
  {cssa_addr(secs,tcs)}
  cssa_addr(secs,tcs) =
  PLUS_va(Secs_baseaddr(secs),
    PLUS_va(LSHIFT_va(Tcs_ossa(tcs), PAGE_SIZE),
          LSHIFT_va(Tcs_cssa(tcs), PAGE_SIZE))));

function pssa_addr(secs_t, tcs_t) : vaddr_t;
axiom (∀ secs: secs_t, tcs: tcs_t ●
  {pssa_addr(secs,tcs)}
  pssa_addr(secs,tcs) =
  MINUS_va(cssa_addr(secs,tcs), LSHIFT_va(k1_vaddr_t, PAGE_SIZE)));

//axiom constraining sha256 to be injective
function sha256(val: int) : sgx_measurement_t;
axiom (∀ val1: int, val2: int ●
  {sha256(val1), sha256(val2)}
  (val1 ≠val2 ⟹sha256(val1) ≠sha256(val2)));

function cmac<a>(k: key_t, x: a) : mactext_t a;
axiom (∀ <a> k: key_t, x1: a, x2: a ●
  {cmac(k,x1), cmac(k,x2)}
  (x1 ≠x2 ⟹cmac(k,x1) ≠cmac(k,x2)));

axiom (∀ val1: int, val2: int ●
  {sha256(val1), sha256(val2)}
  (val1 ≠val2 ⟹sha256(val1) ≠sha256(val2)));

//axiom constraining chained sha256 to be injective
function sha256update(prev: sgx_measurement_t, update: int) : sgx_measurement_t;
axiom (∀ prev1: sgx_measurement_t, update1: int, prev2: sgx_measurement_t, update2: int ●
  {sha256update(prev1,update1), sha256update(prev2,update2)}
  (prev1 ≠prev2 ∨update1 ≠update2) ⟹
  (sha256update(prev1,update1) ≠sha256update(prev2,update2)));

//injective axiom for hash (we use hash only when its unclear which hash algorithm Intel is using)
function hash<a>(x: a) : hashtext_t a;
axiom (∀ <a> x1: a, x2: a ●{hash(x1), hash(x2)}
      (x1 ≠x2 ⟹hash(x1) ≠hash(x2)));

function derive_key_ereport(attributes: attributes_t, measurement: sgx_measurement_t): key_t;
function derive_key_egetkey(keyname: keyname_t,
                        isvprodid: int, isvsvn: int, attributes: attributes_t,
                        mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t): key_t;
axiom (∀ attr1: attributes_t, meas1: sgx_measurement_t,
            keyname: keyname_t, isvprodid: int, isvsvn: int,
            attr2: attributes_t, mrenclave: sgx_measurement_t, mrsigner: hashtext_t key_t ●
            {derive_key_ereport(attr1, meas1), derive_key_egetkey(keyname, isvprodid, isvsvn, attr2, mrenclave, mrsigner)}
            (attr1 =attr2 ∧meas1 =mrenclave ∧keyname =report_key) ⟹
            (derive_key_ereport(attr1, meas1) =derive_key_egetkey(keyname, isvprodid, isvsvn, attr2, mrenclave, mrsigner)));


function decrypt<a>(key: key_t, ciphertext: ciphertext_t a): a;
function encrypt<a>(key: key_t, p: a): ciphertext_t a;
axiom (∀ <a> k: key_t, c: ciphertext_t a ●{decrypt(k, c)} encrypt(k, decrypt(k, c)) =c);
axiom (∀ <a> k: key_t, p: a ●{encrypt(k,p)} decrypt(k, encrypt(k, p)) =p);

//concatenation must be injective
function concat_two_int_to_one(fst: int, snd: int) : int;
axiom (∀ fst1: int, fst2: int, snd1: int, snd2: int ●
  {concat_two_int_to_one(fst1, snd1), concat_two_int_to_one(fst2, snd2)}
  (fst1 ≠fst2 ∨snd1 ≠snd2) ⟹(concat_two_int_to_one(fst1,snd1) ≠concat_two_int_to_one(fst2,snd2)));

function vaddr_to_int(v: vaddr_t) : int;

//cast a regular page to an int
function reg_to_int(val: word_t) : int;
//would like the cast to be one-to-one
axiom (∀ val1: word_t, val2: word_t ●
  {reg_to_int(val1), reg_to_int(val2)}
  (val1 ≠val2 ⟹reg_to_int(val1) ≠reg_to_int(val2)));
```

```
//cast a regular page to an int
function tcs_to_int(val: tcs_t) : int;
//would like the cast to be one-to-one
axiom (∀ val1: tcs_t, val2: tcs_t ●
  {tcs_to_int(val1), tcs_to_int(val2)}
  (val1 ≠val2 ⟹tcs_to_int(val1) ≠tcs_to_int(val2)));


//******************** SGX Instructions ********************

procedure {: inline 1} ecreate_unchecked(la: vaddr_t, secs: secs_t)
modifies epcm, mem_secs, mem_reg, mem_tcs, arbitrary_write_count;
{
  var pa: wap_addr_t;
  var measurement : sgx_measurement_t; //computing value for mrenclave
  var ssaframesize : int;

  pa :=page_table_map[la];

  ssaframesize :=1;

  //valid epcm of type secs and enclave address of 0, enclavesecs undefined thus set to 0
  //arg1: valid, arg2-4: rwx, arg5: page type, arg6: enclavesecs, arg7: enclaveaddress
  epcm[pageof_pa(pa)] :=Epcm(true, false, false, false, pt_secs, k0_wap_addr_t, k0_vaddr_t);

  //set baseaddr and size from the input secs struct, initialized must be false, mrsigner is not yet set
  call unchecked_write_secs(pageof_pa(pa),        //writing to secs[pa]
                        Secs(Secs_baseaddr(secs), //baseaddr (evrange base)
                        Secs_size(secs),          //size (evrange high - evrange base)
                        false,                    //initialized
                        measurement,              //value doesn't matter for this model
                        hash(dummy_signing_key), //value doesn't matter for this model
                        0,                        //ISV product id
                        0,                        //ISV version number
                        Secs_attributes(secs))); //Attributes
}

//la: pagetable[la] holds the secs page of this enclave
//secs: secs struct to populate
procedure {: inline 1} ecreate(la: vaddr_t, secs: secs_t)
returns (result: sgx_api_result_t)
modifies epcm, mem_secs, mem_reg, mem_tcs, arbitrary_write_count;
{
  var pa: wap_addr_t;
  pa :=page_table_map[la];

  if (pageof_va(la) ≠la) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (¬page_table_valid[la]) { result :=sgx_api_invalid_value; return; }
  if (¬is_epc_address(pa) ∨Epcm_valid(epcm[pageof_pa(pa)])) { result :=sgx_api_invalid_value; return; } //must be a free epc address
  if (Core_state_cr_enclave_mode(core_state[curr_core])) { result :=sgx_api_invalid_value; return; } //enclave cannot call ecreate
  if (¬ (GT_va(Secs_size(secs), k0_vaddr_t)) ) { result :=sgx_api_invalid_value; return; } //positive sized enclave

  call ecreate_unchecked(la, secs);
  result :=sgx_api_success;
}

//rcx: address of destination epc page
//rbx_linaddr: linear address with which one addresses this epc page
//rbx_secs: linear address of the SECS page
//d: data to write to the epc page
procedure {: inline 1} eadd_unchecked(rbx_linaddr: vaddr_t, rbx_secs: vaddr_t, rcx: vaddr_t, r: bool, w: bool, x: bool, d: mem_t)
modifies epcm, mem_secs, mem_reg, mem_tcs, arbitrary_write_count;
{
  var epc_pa : wap_addr_t;
  var secs_pa : wap_addr_t;

  epc_pa :=page_table_map[rcx];
  secs_pa :=page_table_map[rbx_secs];
  epcm[pageof_pa(epc_pa)] :=Epcm(true, r, w, x, pt_reg, secs_pa, pageof_va(rbx_linaddr));
  havoc mem_reg;
  assume (∀ a: wap_addr_t ●((LT_wapa(a, epc_pa) ∧GE_wapa(a, PLUS_wapa(epc_pa, 4096bv22))) ⟹mem_reg[a] =old(mem_reg)[a]) ∧
                          ((GE_wapa(a, epc_pa) ∧LT_wapa(a, PLUS_wapa(epc_pa, 4096bv22))) ⟹mem_reg[a] =d[a]));
  mem_secs[pageof_pa(epc_pa)] :=arbitrary_secs_val(arbitrary_write_count);
  mem_tcs[pageof_pa(epc_pa)] :=arbitrary_tcs_val(arbitrary_write_count);
  arbitrary_write_count :=arbitrary_write_count + 1;
}

procedure {: inline 1} eadd(rbx_linaddr: vaddr_t, rbx_secs: vaddr_t, rcx: vaddr_t, r: bool, w: bool, x: bool, d: mem_t)
returns (result: sgx_api_result_t)
modifies epcm, mem_secs, mem_reg, mem_tcs, arbitrary_write_count;
{
  var epc_pa : wap_addr_t;
  var secs_pa : wap_addr_t;

  epc_pa :=page_table_map[rcx];
```

```
  secs_pa :=page_table_map[rbx_secs];

  if (pageof_va(rcx) ≠rcx) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (pageof_va(rbx_secs) ≠rbx_secs) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (pageof_pa(epc_pa) ≠epc_pa) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (pageof_pa(secs_pa) ≠secs_pa) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (¬page_table_valid[rcx]) { result :=sgx_api_invalid_value; return; }
  if (¬page_table_valid[rbx_secs]) { result :=sgx_api_invalid_value; return; }
  if ( ¬(is_epc_address(secs_pa) ∧
         Epcm_valid(epcm[pageof_pa(secs_pa)]) ∧
         Epcm_pt(epcm[pageof_pa(secs_pa)]) =pt_secs ∧
         ¬Secs_initialized(mem_secs[secs_pa])) ) { result :=sgx_api_invalid_value; return; }
  if ( ¬(GE_va(rbx_linaddr, Secs_baseaddr(mem_secs[secs_pa])) ∧
        LT_va(rbx_linaddr, PLUS_va(Secs_baseaddr(mem_secs[secs_pa]), Secs_size(mem_secs[secs_pa])))) ) { result :=sgx_api_invalid_value;
             return; }
  if ( ¬(is_epc_address(epc_pa) ∧(¬Epcm_valid(epcm[pageof_pa(epc_pa)])))) ) { result :=sgx_api_invalid_value; return; } //must be a free epc
        address
  if (Core_state_cr_enclave_mode(core_state[curr_core])) { result :=sgx_api_invalid_value; return; } //enclave cannot call eadd

  call eadd_unchecked(rbx_linaddr, rbx_secs, rcx, r, w, x, d);
  result :=sgx_api_success;
}

// remove EPC page at EPC address la
procedure {: inline 1} eremove_unchecked(rcx: vaddr_t)
modifies epcm;
{
  var epc_pa: wap_addr_t;
  epc_pa :=page_table_map[rcx];
  //set valid bit to false, which dummy_epcm has
  epcm[pageof_pa(epc_pa)] :=dummy_epcm;
}

procedure {: inline 1} eremove(rcx: vaddr_t)
returns (result: sgx_api_result_t)
modifies epcm;
{
  var epc_pa : wap_addr_t;
  epc_pa :=page_table_map[rcx];

  if (pageof_va(rcx) ≠rcx) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (pageof_pa(epc_pa) ≠epc_pa) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (¬page_table_valid[rcx]) { result :=sgx_api_invalid_value; return; }
  if (¬ (is_epc_address(epc_pa)) ) { result :=sgx_api_invalid_value; return; }
  if (¬ ((Epcm_valid(epcm[pageof_pa(epc_pa)]) ∧
          Epcm_pt(epcm[pageof_pa(epc_pa)]) =pt_secs) ⟹
          no_pages_in_enclave(epcm, epc_pa)) ) { result :=sgx_api_invalid_value; return; }
  if (¬ ((Epcm_valid(epcm[pageof_pa(epc_pa)]) ∧
          Epcm_pt(epcm[pageof_pa(epc_pa)]) ≠pt_secs) ⟹
          no_threads_in_enclave(core_state, Epcm_enclavesecs(epcm[pageof_pa(epc_pa)]))) ) { result :=sgx_api_invalid_value; return; }
  if (Core_state_cr_enclave_mode(core_state[curr_core])) { result :=sgx_api_invalid_value; return; }

  call eremove_unchecked(rcx);
  result :=sgx_api_success;
}

// take measurement of a 256 byte region, must be invokes 16 times to measure a page
// However, this model takes the entire measurement of the page at once
procedure {: inline 1} eextend_unchecked(rcx: vaddr_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs;
{
  var secs : secs_t;
  var tmp_enclaveoffset : vaddr_t;
  var tmp_mrenclave : sgx_measurement_t;
  var epc_pa : wap_addr_t;

  epc_pa :=page_table_map[rcx];
  secs :=mem_secs[Epcm_enclavesecs(epcm[pageof_pa(epc_pa)])];

  tmp_enclaveoffset :=MINUS_va(Epcm_enclaveaddress(epcm[pageof_pa(epc_pa)]), Secs_baseaddr(secs));
  tmp_enclaveoffset :=PLUS_va(tmp_enclaveoffset, (0bv20 ++ rcx[12: 0]));

  tmp_mrenclave :=Secs_mrenclave(secs);
  tmp_mrenclave :=sha256update(tmp_mrenclave, vaddr_to_int(tmp_enclaveoffset));
  if (Epcm_pt(epcm[pageof_pa(epc_pa)]) =pt_reg) {
    tmp_mrenclave :=sha256update(tmp_mrenclave, reg_to_int(mem_reg[epc_pa]));
  } else {
    tmp_mrenclave :=sha256update(tmp_mrenclave, tcs_to_int(mem_tcs[epc_pa]));
  }

  call unchecked_write_secs(Epcm_enclavesecs(epcm[pageof_pa(epc_pa)]),
                        Secs(Secs_baseaddr(secs), Secs_size(secs), Secs_initialized(secs),
                        tmp_mrenclave, Secs_mrsigner(secs),
                        Secs_isvprodid(secs), Secs_isvsvn(secs), Secs_attributes(secs)));
}
```

```
procedure {: inline 1} eextend(rcx: vaddr_t)
returns (result: sgx_api_result_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs;
{
  var epc_pa : wap_addr_t;
  epc_pa :=page_table_map[rcx];

  if (¬page_table_valid[rcx]) { result :=sgx_api_invalid_value; return; }
  if (¬ (is_epc_address(epc_pa)) ) { result :=sgx_api_invalid_value; return; }
  if (¬ (Epcm_valid(epcm[pageof_pa(epc_pa)]) ∧
        (Epcm_pt(epcm[pageof_pa(epc_pa)]) =pt_reg ∨
         Epcm_pt(epcm[pageof_pa(epc_pa)]) =pt_tcs)) ) { result :=sgx_api_invalid_value; return; }
  if (Secs_initialized(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(epc_pa)])])) { result :=sgx_api_invalid_value; return; }
  if (Core_state_cr_enclave_mode(core_state[curr_core])) { result :=sgx_api_invalid_value; return; }

  call eextend_unchecked(rcx);
  result :=sgx_api_success;
}

//rbx: targetinfo struct, rcx: reportdata struct, rdx: addr containing output report struct
procedure {: inline 1} ereport_unchecked(targetinfo: targetinfo_t, reportdata: word_t)
returns (report_maced: report_maced_t)
{
  var tmp_reportkey : key_t;
  var tmp_currentsecs : secs_t;
  var report : report_t;
  var tmp_report_mac : mactext_t report_t;

  tmp_reportkey :=derive_key_ereport(Targetinfo_attributes(targetinfo),
                                     Targetinfo_measurement(targetinfo));
  tmp_currentsecs :=mem_secs[Core_state_cr_active_secs(core_state[curr_core])];
  report :=Report(Secs_isvprodid(tmp_currentsecs),
                  Secs_isvsvn(tmp_currentsecs),
                  Secs_attributes(tmp_currentsecs),
                  reportdata,
                  Secs_mrenclave(tmp_currentsecs),
                  Secs_mrsigner(tmp_currentsecs));

  tmp_report_mac :=cmac(tmp_reportkey, report);
  report_maced :=Report_maced(report, tmp_report_mac);
}

procedure {: inline 1} ereport(targetinfo: targetinfo_t, reportdata: word_t)
returns (report_maced: report_maced_t, result: sgx_api_result_t)
{
  if (¬ (Core_state_cr_enclave_mode(core_state[curr_core]))) { result :=sgx_api_invalid_value; return; }
  call report_maced :=ereport_unchecked(targetinfo, reportdata);
  result :=sgx_api_success;
}

// initialize enclave whose SECS page is located at EPC address ls
//rbx -> sigstruct, rcx -> secs, rdx -> einittoken
procedure {: inline 1} einit_unchecked(sigstruct_signed: sigstruct_signed_t, rcx: vaddr_t, einittoken: einittoken_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs;
{
  var secs : secs_t;
  var tmp_mrsigner : hashtext_t key_t;
  var sigstruct : sigstruct_t;
  var secs_pa : wap_addr_t;

  secs_pa :=page_table_map[rcx];

  sigstruct :=Sigstruct_signed_sigstruct(sigstruct_signed);
  tmp_mrsigner :=hash(Sigstruct_modulus(sigstruct));

  secs :=mem_secs[secs_pa];

  call unchecked_write_secs(secs_pa,
                        Secs(Secs_baseaddr(secs), Secs_size(secs), true,
                             Secs_mrenclave(secs), tmp_mrsigner,
                             Sigstruct_isvprodid(sigstruct), Sigstruct_isvsvn(sigstruct),
                             Secs_attributes(secs)));
}

procedure {: inline 1} einit(sigstruct_signed: sigstruct_signed_t, rcx: vaddr_t, einittoken: einittoken_t)
returns (result: sgx_api_result_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs;
{
  var secs_pa : wap_addr_t;
  secs_pa :=page_table_map[rcx];

  if (pageof_va(rcx) ≠rcx) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (pageof_pa(secs_pa) ≠secs_pa) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (¬page_table_valid[rcx]) { result :=sgx_api_invalid_value; return; }
  if (¬ (Epcm_valid(epcm[pageof_pa(secs_pa)]) ∧Epcm_pt(epcm[pageof_pa(secs_pa)]) =pt_secs) ) { result :=sgx_api_invalid_value; return; }
  if (¬ (is_epc_address(secs_pa) ∧¬Secs_initialized(mem_secs[secs_pa])) ) { result :=sgx_api_invalid_value; return; }
```

```
  if (Core_state_cr_enclave_mode(core_state[curr_core])) { result :=sgx_api_invalid_value; return; } //enclave cannot call einit

  call einit_unchecked(sigstruct_signed, rcx, einittoken);
  result :=sgx_api_success;
}


//Enter an enclave via a thread whose TCS lives at EPC address la
procedure {: inline 1} eenter_unchecked(rbx: vaddr_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs, core_state;
{
  var tcs_pa: wap_addr_t;
  var secs_pa: wap_addr_t;
  var pcssa: wap_addr_t;

  tcs_pa :=page_table_map[rbx];
  secs_pa :=Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)]);
  pcssa :=page_table_map[cssa_addr(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])], mem_tcs[tcs_pa])];

  call unchecked_write_tcs(tcs_pa, Tcs(true,
                               Tcs_interrupted(mem_tcs[tcs_pa]),
                               Tcs_ossa(mem_tcs[tcs_pa]),
                               Tcs_nssa(mem_tcs[tcs_pa]),
                               Tcs_cssa(mem_tcs[tcs_pa])
                              ));

  //Core_state(cr_enclave_mode: bool, cr_tcs_pa: wap_addr_t, cr_active_secs: wap_addr_t, cr_elrange: lr_register_t, ssa_pa: wap_addr_t)
  core_state[curr_core] :=Core_state(true, //cr_enclave_mode
                           tcs_pa, //TCS pa
                           secs_pa, //SECS pa
                           Lr_register(Secs_baseaddr(mem_secs[secs_pa]), Secs_size(mem_secs[secs_pa])), //evrange base and size
                           pcssa
                          );
}

procedure {: inline 1} eenter(rbx: vaddr_t)
returns (result: sgx_api_result_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs, core_state;
{
  var tcs_pa: wap_addr_t;
  tcs_pa :=page_table_map[rbx];

  if (pageof_va(rbx) ≠rbx) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (pageof_pa(tcs_pa) ≠tcs_pa) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (¬page_table_valid[rbx]) { result :=sgx_api_invalid_value; return; }
  if (¬is_epc_address(tcs_pa)) { result :=sgx_api_invalid_value; return; }
  if (¬ (¬Tcs_active(mem_tcs[tcs_pa]) ∧¬Tcs_interrupted(mem_tcs[tcs_pa])) ) { result :=sgx_api_invalid_value; return; }
  if (¬ (Epcm_enclaveaddress(epcm[pageof_pa(tcs_pa)]) =rbx ∧Epcm_pt(epcm[pageof_pa(tcs_pa)]) =pt_tcs ∧Epcm_valid(epcm[pageof_pa(tcs_pa)]))
      ) { result :=sgx_api_invalid_value; return; }
  if (¬ (LT_va(Tcs_cssa(mem_tcs[tcs_pa]), Tcs_nssa(mem_tcs[tcs_pa])) ∧
        is_epc_address(page_table_map[cssa_addr(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])], mem_tcs[tcs_pa])]) ∧
        Epcm_valid(epcm[pageof_pa(page_table_map[cssa_addr(mem_secs[Epcm_enclavesecs(epcm[tcs_pa])], mem_tcs[tcs_pa])])]) ∧
        Epcm_pt(epcm[pageof_pa(page_table_map[cssa_addr(mem_secs[Epcm_enclavesecs(epcm[tcs_pa])], mem_tcs[tcs_pa])])]) =pt_reg ∧
        Epcm_enclaveaddress(epcm[pageof_pa(page_table_map[cssa_addr(mem_secs[Epcm_enclavesecs(epcm[tcs_pa])], mem_tcs[tcs_pa])])]) =
              cssa_addr(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])], mem_tcs[tcs_pa]) ∧
        Epcm_enclavesecs(epcm[pageof_pa(page_table_map[cssa_addr(mem_secs[Epcm_enclavesecs(epcm[tcs_pa])], mem_tcs[tcs_pa])])]) =
              Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)]) ∧
        Secs_initialized(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])])) ) { result :=sgx_api_invalid_value; return; }
  if (Core_state_cr_enclave_mode(core_state[curr_core])) { result :=sgx_api_invalid_value; return; }

  call eenter_unchecked(rbx);
  result :=sgx_api_success;
}

//Resume an enclave via a thread whose TCS lives at la
procedure {: inline 1} eresume_unchecked(rbx: vaddr_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs, core_state, gpregs;
{
  var tcs_pa: wap_addr_t;
  var secs_pa: wap_addr_t;
  var ppssa: wap_addr_t;

  tcs_pa :=page_table_map[rbx];
  secs_pa :=Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)]);
  ppssa :=page_table_map[pssa_addr(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])], mem_tcs[tcs_pa])];

  call unchecked_write_tcs(tcs_pa, Tcs(true, //active
                               false, //interrupted
                               Tcs_ossa(mem_tcs[tcs_pa]),
                               Tcs_nssa(mem_tcs[tcs_pa]),
                               MINUS_va(Tcs_cssa(mem_tcs[tcs_pa]), k1_vaddr_t)));

  core_state[curr_core] :=Core_state(true,
                           tcs_pa,
                           secs_pa,
                           Lr_register(Secs_baseaddr(mem_secs[secs_pa]), Secs_size(mem_secs[secs_pa])),
```

```
                            ppssa
                           );
  gpregs[curr_core] :=word_to_gpregs(mem_reg[ppssa]);
}

procedure {: inline 1} eresume(rbx: vaddr_t)
returns (result: sgx_api_result_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs, core_state, gpregs;
{
  var tcs_pa: wap_addr_t;
  tcs_pa :=page_table_map[rbx];

  if (pageof_va(rbx) ≠rbx) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (pageof_pa(tcs_pa) ≠tcs_pa) { result :=sgx_api_invalid_value; return; } //la must be aligned
  if (¬page_table_valid[rbx]) { result :=sgx_api_invalid_value; return; }
  if (Tcs_active(mem_tcs[tcs_pa])) { result :=sgx_api_invalid_value; return; }
  if (¬is_epc_address(tcs_pa)) { result :=sgx_api_invalid_value; return; }
  if (¬ (Epcm_valid(epcm[pageof_pa(tcs_pa)])) ) { result :=sgx_api_invalid_value; return; }
  if (¬ (Epcm_enclaveaddress(epcm[pageof_pa(tcs_pa)]) =rbx) ∧(Epcm_pt(epcm[pageof_pa(tcs_pa)]) =pt_tcs) ) { result :=
       sgx_api_invalid_value; return; }
  if (¬ (GT_va(Tcs_cssa(mem_tcs[tcs_pa]), k0_vaddr_t)) ) { result :=sgx_api_invalid_value; return; }
  if (¬ (is_epc_address(page_table_map[pssa_addr(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])], mem_tcs[tcs_pa])]) ∧
       Epcm_valid(epcm[pageof_pa(page_table_map[pssa_addr(mem_secs[Epcm_enclavesecs(epcm[tcs_pa])], mem_tcs[tcs_pa])])]) ∧
       Epcm_pt(epcm[pageof_pa(page_table_map[pssa_addr(mem_secs[Epcm_enclavesecs(epcm[tcs_pa])], mem_tcs[tcs_pa])])]) =pt_reg) ) { }
  if (¬ (Epcm_enclaveaddress(epcm[pageof_pa(page_table_map[pssa_addr(mem_secs[Epcm_enclavesecs(epcm[tcs_pa])], mem_tcs[tcs_pa])])]) =
       pssa_addr(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])], mem_tcs[tcs_pa])) ) { result :=sgx_api_invalid_value; return; }
  if (¬ (Epcm_enclavesecs(epcm[pageof_pa(page_table_map[pssa_addr(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])], mem_tcs[tcs_pa])])]) =
       Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])) ) { result :=sgx_api_invalid_value; return; }
  if (¬ (Secs_initialized(mem_secs[Epcm_enclavesecs(epcm[pageof_pa(tcs_pa)])])) ) { result :=sgx_api_invalid_value; return; }
  if (Core_state_cr_enclave_mode(core_state[curr_core])) { result :=sgx_api_invalid_value; return; }

  call eresume_unchecked(rbx);
  result :=sgx_api_success;
}

function AES_GCM_ENC_reg(plaintext: word_t) : word_t;
function AES_GCM_DEC_reg(ciphertext: word_t) : word_t;
axiom (∀ ptxt: word_t, ctxt: word_t ●{AES_GCM_ENC_reg(ptxt), AES_GCM_DEC_reg(ctxt)}
            AES_GCM_ENC_reg(ptxt) =ctxt ⟹AES_GCM_DEC_reg(ctxt) =ptxt);
axiom (∀ ptxt: word_t, ctxt: word_t ●{AES_GCM_ENC_reg(ptxt), AES_GCM_DEC_reg(ctxt)}
            AES_GCM_DEC_reg(ctxt) =ptxt ⟹AES_GCM_ENC_reg(ptxt) =ctxt);

function AES_GCM_ENC_tcs(plaintext: tcs_t) : word_t;
function AES_GCM_DEC_tcs(ciphertext: word_t) : tcs_t;
axiom (∀ ptxt: tcs_t, ctxt: word_t ●{AES_GCM_ENC_tcs(ptxt), AES_GCM_DEC_tcs(ctxt)}
            AES_GCM_ENC_tcs(ptxt) =ctxt ⟹AES_GCM_DEC_tcs(ctxt) =ptxt);
axiom (∀ ptxt: tcs_t, ctxt: word_t ●{AES_GCM_ENC_tcs(ptxt), AES_GCM_DEC_tcs(ctxt)}
            AES_GCM_DEC_tcs(ctxt) =ptxt ⟹AES_GCM_ENC_tcs(ptxt) =ctxt);

function AES_GCM_ENC_secs(plaintext: secs_t) : word_t;
function AES_GCM_DEC_secs(ciphertext: word_t) : secs_t;
axiom (∀ ptxt: secs_t, ctxt: word_t ●{AES_GCM_ENC_secs(ptxt), AES_GCM_DEC_secs(ctxt)}
            AES_GCM_ENC_secs(ptxt) =ctxt ⟹AES_GCM_DEC_secs(ctxt) =ptxt);
axiom (∀ ptxt: secs_t, ctxt: word_t ●{AES_GCM_ENC_secs(ptxt), AES_GCM_DEC_secs(ctxt)}
            AES_GCM_DEC_secs(ctxt) =ptxt ⟹AES_GCM_ENC_secs(ptxt) =ctxt);


//copy a page from EPC page la to dst_la in unprotected memory
procedure {: inline 1} ewb(la: vaddr_t, pageinfo: pageinfo_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs, epcm;
requires (¬Core_state_cr_enclave_mode(core_state[curr_core])); //only OS/VMM allowed to call ewb
requires (is_epc_address(page_table_map[la]) ∧Epcm_valid(epcm[page_table_map[la]]));
requires ((Epcm_pt(epcm[page_table_map[la]]) =pt_reg) ∨
          (Epcm_pt(epcm[page_table_map[la]]) =pt_tcs)) ⟹
          no_threads_in_enclave(core_state, Epcm_enclavesecs(epcm[page_table_map[la]]));
requires (Epcm_pt(epcm[page_table_map[la]]) =pt_secs) ⟹
         (no_threads_in_enclave(core_state, page_table_map[la]) ∧
          no_pages_in_enclave(epcm, page_table_map[la]));
requires (¬is_epc_address(page_table_map[Pageinfo_srcpge(pageinfo)]));
{
  var pa: wap_addr_t;
  var tmp_srcpge : vaddr_t;

  pa :=page_table_map[la];
  tmp_srcpge :=Pageinfo_srcpge(pageinfo);

  if (Epcm_pt(epcm[pa]) =pt_reg) {
    call unchecked_write_reg(page_table_map[tmp_srcpge], AES_GCM_ENC_reg(mem_reg[pa]));
  } else if (Epcm_pt(epcm[pa]) =pt_tcs) {
    call unchecked_write_reg(page_table_map[tmp_srcpge], AES_GCM_ENC_tcs(mem_tcs[pa]));
  } else if (Epcm_pt(epcm[pa]) =pt_secs) {
    call unchecked_write_reg(page_table_map[tmp_srcpge], AES_GCM_ENC_secs(mem_secs[pa]));
  }

  epcm[pa] :=dummy_epcm;
}
```

```
//copy a page from src_la to EPC page la
procedure {: inline 1} eldu(la: vaddr_t, pageinfo: pageinfo_t)
modifies mem_reg, mem_tcs, mem_secs, epcm, arbitrary_write_count;
requires (is_epc_address(page_table_map[la]) ∧¬Epcm_valid(epcm[page_table_map[la]]));
requires ((Secinfo_flags_pt(Pageinfo_secinfo(pageinfo)) =pt_reg) ∨
         (Secinfo_flags_pt(Pageinfo_secinfo(pageinfo)) =pt_tcs)) ⟹
         (is_epc_address(Pageinfo_secs(pageinfo)) ∧
          Epcm_valid(epcm[Pageinfo_secs(pageinfo)]) ∧
          Epcm_pt(epcm[Pageinfo_secs(pageinfo)]) =pt_secs);
requires (Secinfo_flags_pt(Pageinfo_secinfo(pageinfo)) =pt_secs) ⟹
        (Pageinfo_secs(pageinfo) =k0_wap_addr_t);
{
  var tmp_srcpge : vaddr_t;
  var tmp_secs : wap_addr_t;
  var tmp_secinfo : secinfo_t;
  var pa: wap_addr_t;
  var tmp_header_secinfo_flags_pt : page_t;

  pa :=page_table_map[la];
  tmp_srcpge :=Pageinfo_srcpge(pageinfo);
  tmp_secs :=Pageinfo_secs(pageinfo);
  tmp_secinfo :=Pageinfo_secinfo(pageinfo);

  tmp_header_secinfo_flags_pt :=Secinfo_flags_pt(tmp_secinfo);
  if (tmp_header_secinfo_flags_pt =pt_reg) {
    call unchecked_write_reg(pa, AES_GCM_DEC_reg(mem_reg[page_table_map[tmp_srcpge]]));
  } else if (tmp_header_secinfo_flags_pt =pt_tcs) {
    call unchecked_write_tcs(pa, AES_GCM_DEC_tcs(mem_reg[page_table_map[tmp_srcpge]]));
  } else if (tmp_header_secinfo_flags_pt =pt_secs) {
    call unchecked_write_secs(pa, AES_GCM_DEC_secs(mem_reg[page_table_map[tmp_srcpge]]));
  }

  epcm[pa] :=Epcm(true, true, true, true,
                  tmp_header_secinfo_flags_pt,
                  Pageinfo_secs(pageinfo),
                  Pageinfo_linaddr(pageinfo));
}

procedure {: inline 1} eexit()
returns (result: sgx_api_result_t)
modifies mem_reg, gpregs, core_state, mem_tcs, mem_secs, arbitrary_write_count;
{
  var ptcs : wap_addr_t;
  if (¬ (Core_state_cr_enclave_mode(core_state[curr_core])) ) { result :=sgx_api_invalid_value; return; }

  ptcs :=Core_state_cr_tcs_pa(core_state[curr_core]);

  core_state[curr_core] :=Core_state(false,
                            Core_state_cr_tcs_pa(core_state[curr_core]),
                            Core_state_cr_active_secs(core_state[curr_core]),
                            Core_state_cr_elrange(core_state[curr_core]),
                            Core_state_ssa_pa(core_state[curr_core])
                           );

  call unchecked_write_tcs(ptcs, Tcs(false,
                                Tcs_interrupted(mem_tcs[ptcs]),
                                Tcs_ossa(mem_tcs[ptcs]),
                                Tcs_nssa(mem_tcs[ptcs]),
                                Tcs_cssa(mem_tcs[ptcs])
                             ));
  result :=sgx_api_success;
}

procedure {: inline 1} aexit(interrupt: bool)
returns (result: sgx_api_result_t)
modifies mem_reg, gpregs, core_state, mem_tcs, mem_secs, arbitrary_write_count;
{
  var pt : wap_addr_t;
  var pc : wap_addr_t;

  if (¬ (Core_state_cr_enclave_mode(core_state[curr_core])) ) { result :=sgx_api_invalid_value; return; }

  pt :=Core_state_cr_tcs_pa(core_state[curr_core]);
  pc :=Core_state_ssa_pa(core_state[curr_core]);

  call unchecked_write_reg(pc, gpregs_to_word(gpregs[curr_core]));
  gpregs[curr_core] :=dummy_gpregs;
  core_state[curr_core] :=Core_state(false,
                            Core_state_cr_tcs_pa(core_state[curr_core]),
                            Core_state_cr_active_secs(core_state[curr_core]),
                            Core_state_cr_elrange(core_state[curr_core]),
                            Core_state_ssa_pa(core_state[curr_core])
                           );
  call unchecked_write_tcs(pt, Tcs(false,
                              interrupt,
```

```
                              Tcs_ossa(mem_tcs[pt]),
                              Tcs_nssa(mem_tcs[pt]),
                              PLUS_va(Tcs_cssa(mem_tcs[pt]), k1_vaddr_t)
                             ));

  result :=sgx_api_success;
}

procedure {: inline 1} interrupt()
returns (result: sgx_api_result_t)
modifies mem_reg, gpregs, core_state, mem_tcs, mem_secs, arbitrary_write_count;
{
  if (Core_state_cr_enclave_mode(core_state[curr_core])) {
    call result :=aexit(true);
  }
}

procedure {: inline 1} exception()
returns (result: sgx_api_result_t)
modifies mem_reg, gpregs, core_state, mem_tcs, mem_secs, arbitrary_write_count;
{
  if (Core_state_cr_enclave_mode(core_state[curr_core])) {
    call result :=aexit(false);
  }
}

//rbx: input keyrequest struct, rcx: output outputdata struct
//return sealing and attestation keys
procedure {: inline 1} egetkey(keyrequest: keyrequest_t) returns (result: key_t)
requires (Core_state_cr_enclave_mode(core_state[curr_core]));
requires (Keyrequest_keyname(keyrequest) =seal_key) ⟹
        (Keyrequest_isvsvn(keyrequest) > Secs_isvsvn(mem_secs[Core_state_cr_active_secs(core_state[curr_core])]));
requires Keyrequest_keyname(keyrequest) =seal_key ∨
        Keyrequest_keyname(keyrequest) =report_key;
{
  var tmp_currentsecs : secs_t;
  var keyname : keyname_t;
  var tmp_mrenclave : sgx_measurement_t;
  var tmp_mrsigner : hashtext_t key_t;
  tmp_currentsecs :=mem_secs[Core_state_cr_active_secs(core_state[curr_core])];
  keyname :=Keyrequest_keyname(keyrequest);

  if (keyname =seal_key) {
    tmp_mrenclave :=0;
    //include enclave identity?
    if (Keyrequest_keypolicy_mrenclave(keyrequest)) {
      tmp_mrenclave :=Secs_mrenclave(tmp_currentsecs);
    }
    tmp_mrsigner :=hash(dummy_signing_key);
    //include enclave author?
    if (Keyrequest_keypolicy_mrsigner(keyrequest)) {
      tmp_mrsigner :=Secs_mrsigner(tmp_currentsecs);
    }
    result :=derive_key_egetkey(seal_key,
                              Secs_isvprodid(tmp_currentsecs),
                              Keyrequest_isvsvn(keyrequest),
                              Secs_attributes(tmp_currentsecs),
                              tmp_mrenclave,
                              tmp_mrsigner);
  }
  else if(keyname =report_key) {
    result :=derive_key_egetkey(report_key,
                              0,
                              0,
                              Secs_attributes(tmp_currentsecs),
                              Secs_mrenclave(tmp_currentsecs),
                              hash(dummy_signing_key));
  }
}


//******************** All other operations ************************

procedure {: inline 1} sgx_store(la: vaddr_t, d: word_t)
returns (result: sgx_api_result_t)
modifies arbitrary_write_count, mem_reg, mem_secs, mem_tcs;
{
  var is_writeable : bool;
  var pa : wap_addr_t;

  call is_writeable :=is_accessible(curr_core, la);
  if (¬is_writeable) { result :=sgx_api_invalid_value; return; }

  call pa :=translate(la);
  if (pa =abort_page) { result :=sgx_api_invalid_value; return; }
```

```
  call unchecked_write_reg(pa, d);
  result :=sgx_api_success;
}

procedure {: inline 1} sgx_load(la: vaddr_t)
returns (d: word_t, result: sgx_api_result_t)
{
  var is_readable : bool;
  var pa : wap_addr_t;

  call is_readable :=is_accessible(curr_core, la);
  if (¬is_readable) { result :=sgx_api_invalid_value; return; }

  call pa :=translate(la);
  if (pa =abort_page) { result :=sgx_api_invalid_value; return; }

  d :=mem_reg[pa];
  result :=sgx_api_success;
}

procedure {: inline 1} switch_thread (core : core_id_t)
modifies curr_core;
{
  curr_core :=core;
}

//Abstract computation
procedure {: inline 1} compute()
modifies gpregs;
{
  var x: gpregs_t;
  gpregs[curr_core] :=x;
}

//Adversary update to page tables
procedure {: inline 1} update_page_table_map(l: vaddr_t, p: wap_addr_t)
modifies page_table_map;
{
  page_table_map[l] :=p;
}
```

# Appendix C

# Model of Sanctum

In this appendix chapter, we present our model of Sanctum in full detail, which was described earlier in Chapter 4. We include the reference implementation (in BoogiePL [14]) of all the operations supported by the Sanctum platform, except the mailbox API used for remote attestation.

```
/*********************************
 * Ghost State                   *
 *********************************/

var owner                          : [region_t] enclave_id_t;
var blocked_at                     : [region_t] int;

var enclave_metadata_valid         : eid2bool_map_t;
var enclave_metadata_is_initialized : eid2bool_map_t;
var enclave_metadata_ev_base       : [enclave_id_t] vaddr_t;
var enclave_metadata_ev_mask       : [enclave_id_t] vaddr_t;
var enclave_metadata_bitmap        : [enclave_id_t] bitmap_t;
var enclave_metadata_load_eptbr    : [enclave_id_t] ppn_t;
var enclave_metadata_dram_region_count : [enclave_id_t] word_t;
var enclave_metadata_last_load_addr : [enclave_id_t] paddr_t;
var enclave_metadata_thread_count  : [enclave_id_t] int;
var enclave_metadata_measurement   : [enclave_id_t] measurement_t;

var thread_metadata_valid          : [thread_id_t] bool;
var thread_metadata_eid            : [thread_id_t] enclave_id_t;
var thread_metadata_entry_pc       : [thread_id_t] vaddr_t;
var thread_metadata_entry_stack    : [thread_id_t] vaddr_t;
var thread_metadata_fault_pc       : [thread_id_t] vaddr_t;
var thread_metadata_fault_stack    : [thread_id_t] vaddr_t;

var os_bitmap                      : bitmap_t;

var os_pc                          : vaddr_t;
var dram_regions_info_block_clock  : int;

/*********************************
 * Constants                     *
 *********************************/

const enclave_metadata_valid_init: eid2bool_map_t;
axiom (∀ i: enclave_id_t ●enclave_metadata_valid_init[i] =false);
const thread_metadata_valid_init: [thread_id_t] bool;
axiom (∀ i: thread_id_t ●thread_metadata_valid_init[i] =false);
const mem_zero : mem_t;
axiom (∀ p : wap_addr_t ●mem_zero[p] =k0_word_t);

const monitor_ok: api_result_t;
axiom monitor_ok =0bv3;
const monitor_invalid_value: api_result_t;
axiom monitor_invalid_value =1bv3;
const monitor_invalid_state: api_result_t;
axiom monitor_invalid_state =2bv3;
const monitor_unknown_error: api_result_t;
axiom monitor_unknown_error =3bv3;
const monitor_access_denied: api_result_t;
axiom monitor_access_denied =4bv3;
```

```
const monitor_unsupported: api_result_t;
axiom monitor_unsupported =5bv3;

/** numbers for each Sanctum API.       **/
const code_api_create_enclave        : int;
const code_api_load_page_table       : int;
const code_api_load_page             : int;
const code_api_assign_dram_region    : int;
const code_api_load_thread           : int;
const code_api_init_enclave          : int;
const code_api_enter_enclave         : int;
const code_api_exit_enclave          : int;
const code_api_block_dram_region     : int;
const code_api_free_dram_region      : int;
const code_api_flush_cached_dram_regions : int;
const code_api_delete_thread         : int;
const code_api_delete_enclave        : int;
// values for each constant.
axiom code_api_create_enclave        =10;
axiom code_api_load_page_table       =20;
axiom code_api_load_page             =30;
axiom code_api_assign_dram_region    =40;
axiom code_api_load_thread           =50;
axiom code_api_init_enclave          =60;
axiom code_api_enter_enclave         =70;
axiom code_api_exit_enclave          =80;
axiom code_api_block_dram_region     =90;
axiom code_api_free_dram_region      =100;
axiom code_api_flush_cached_dram_regions =110;
axiom code_api_delete_thread         =120;
axiom code_api_delete_enclave        =130;
/** end numbers for each Sanctum API.   **/

function {: inline} assigned(r: enclave_id_t) : bool
  { r ≠null_enclave_id ∧r ≠free_enclave_id ∧r ≠blocked_enclave_id ∧r ≠metadata_enclave_id } //can the region be freed and reassigned?

/*********************************
 * Sanctum Monitor APIs        *
 *********************************/
procedure initialize_sanctum();
 modifies cpu_evbase,
         cpu_evmask,
         cpu_eptbr,
         cpu_ptbr,
         cpu_drbmap,
         cpu_edrbmap,
         cpu_parbase,
         cpu_eparbase,
         cpu_parmask,
         cpu_eparmask,
         cpu_dmarbase,
         cpu_dmarmask,
         owner,
         mem,
         core_info_enclave_id,
         core_info_thread_id,
         enclave_metadata_valid,
         enclave_metadata_is_initialized,
         thread_metadata_valid,
         os_bitmap;
ensures cpu_evbase =k0_vaddr_t;
ensures cpu_evmask =k0_vaddr_t;
ensures cpu_eptbr =k0_ppn_t;
ensures cpu_ptbr =k2_ppn_t;
ensures cpu_drbmap =k1_bitmap_t;
ensures cpu_edrbmap =k0_bitmap_t;
ensures cpu_parbase =0bv9 ++ 0bv3 ++ 0bv12;
ensures cpu_parmask =0bv9 ++ 0bv3 ++ 255bv12;
ensures cpu_eparbase =cpu_parbase;
ensures cpu_eparmask =cpu_parmask;
ensures cpu_dmarbase =0bv9 ++ 0bv3 ++ 256bv12;
ensures cpu_dmarmask =0bv9 ++ 0bv3 ++ 255bv12;
// most regions free.
ensures (∀ r : region_t ●
           if r =k0_region_t
               then owner[r] =null_enclave_id
               else owner[r] =free_enclave_id);
// no enclave executing.
ensures core_info_enclave_id =null_enclave_id;
ensures os_bitmap =1bv8;
// no enclaves.
ensures (∀ e : enclave_id_t ●¬enclave_metadata_valid[e]);
ensures (∀ e : enclave_id_t ●¬enclave_metadata_is_initialized[e]);
// no threads.
ensures (∀ t : thread_id_t ●¬thread_metadata_valid[t]);
// mem zero.
```

```
ensures (∀ p : wap_addr_t • mem[p] =k0_word_t);


procedure {: inline 1} create_metadata_region(region: region_t)
  returns (result: api_result_t)
  modifies owner;
{
  //must be called by OS
  if (core_info_enclave_id ≠null_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  if (¬is_valid_dram_region(region)) {
    result :=monitor_invalid_value; return;
  }

  //can only convert a free region into a metadata region
  if (owner[region] ≠free_enclave_id) {
    result :=monitor_invalid_state; return;
  }

  owner[region] :=metadata_enclave_id;

  result :=monitor_ok; return;
}

procedure {: inline 1} create_enclave(eid: enclave_id_t, evbase: vaddr_t, evmask: vaddr_t)
  returns (result: api_result_t)
  modifies enclave_metadata_valid,
           enclave_metadata_is_initialized,
           enclave_metadata_ev_base,
           enclave_metadata_ev_mask,
           enclave_metadata_bitmap,
           enclave_metadata_thread_count,
           enclave_metadata_load_eptbr,
           enclave_metadata_dram_region_count,
           enclave_metadata_last_load_addr,
           enclave_metadata_measurement;
{
  var dram_region : region_t;
  var measurement : measurement_t;

  //must be called by OS
  if (core_info_enclave_id ≠null_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  if (¬is_valid_range_va(evbase, evmask)) {
    result :=monitor_invalid_value; return;
  }

  //enclave must get at least a page of virtual address space
  if (LT_va(PLUS_va(evmask, k1_vaddr_t), kPGSZ_vaddr_t)) {
    result :=monitor_invalid_value; return;
  }

  //metadata must live within DRAM
  if (¬is_dram_address(eid) ∨¬is_page_aligned_pa(eid)) {
    result :=monitor_invalid_value; return;
  }

  dram_region :=dram_region_for(eid);
  if (owner[dram_region] ≠metadata_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  /* BUG: missing check */
  if (enclave_metadata_valid[eid]) {
    result :=monitor_invalid_state; return;
  }

  /* BUG: missing check */
  if (¬assigned(eid)) {
    result :=monitor_invalid_value; return;
  }

  enclave_metadata_valid[eid] :=true;
  enclave_metadata_thread_count[eid] :=0;
  enclave_metadata_is_initialized[eid] :=false;
  enclave_metadata_ev_base[eid] :=evbase;
  enclave_metadata_ev_mask[eid] :=evmask;
  enclave_metadata_bitmap[eid] :=k0_bitmap_t;
  enclave_metadata_load_eptbr[eid] :=k0_ppn_t;
  enclave_metadata_dram_region_count[eid] :=k0_word_t;
  enclave_metadata_last_load_addr[eid] :=k0_paddr_t;
```

```
  /* do the measurement. */
  measurement :=code_api_create_enclave;
  measurement :=update_digest(enclave_metadata_thread_count[eid], measurement);
  measurement :=update_digest(vaddr2int(evbase), measurement);
  measurement :=update_digest(vaddr2int(evmask), measurement);
  enclave_metadata_measurement[eid] :=measurement;

  result :=monitor_ok; return;
}

procedure _clear_mapped_pages(ptbr : ppn_t);
  modifies ptbl_acl_map;
  ensures (∀ p : ppn_t, v : vpn_t •
      if p =ptbr
        then ptbl_acl_map[p, v] =k_pg_invalid_acl
        else ptbl_acl_map[p, v] =old(ptbl_acl_map)[p, v]);


procedure _clear_page(ppn : ppn_t);
  modifies mem;
  ensures (∀ pa : wap_addr_t •
             if wpaddr2ppn(pa) =ppn
                 then mem[pa] =k0_word_t
                 else mem[pa] =old(mem)[pa]);

procedure {: inline 1} load_page_table(eid: enclave_id_t, vaddr: vaddr_t, paddr: paddr_t, acl: pte_acl_t, level: int)
  returns (result: api_result_t)
  modifies enclave_metadata_load_eptbr, cpu_ptbr;
  modifies enclave_metadata_measurement;
  modifies ptbl_addr_map, ptbl_acl_map;

  ensures ((core_info_enclave_id ≠null_enclave_id)                              ∨
          (level < 0 ∨level > 2)                                                ∨
           ¬is_dram_address(paddr)                                               ∨
           ¬is_page_aligned_pa(paddr)                                            ∨
           (eid ≠null_enclave_id ∧¬is_valid_enclave_id(enclave_metadata_valid, eid)) ∨
           (eid =null_enclave_id ∧level ≠2)                                    ∨
           (enclave_metadata_is_initialized[eid])                               ∨
           (owner[dram_region_for(paddr)] ≠eid ∧level > 0)                     ∨
           (level ≠2 ∧
           ((eid =null_enclave_id ∧cpu_ptbr =k0_ppn_t) ∨
            (eid ≠null_enclave_id ∧enclave_metadata_load_eptbr[eid] =k0_ppn_t))))
           ⟹(result ≠monitor_ok);
{
  var pte: word_t;
  var ptbr: ppn_t;
  var eptbr_p: ppn_t;
  var paddr_region: region_t;
  var eid_region: region_t;
  var success: bool;
  var measurement : measurement_t;
  var new_ptbl_acl_map : ptbl_acl_map_t;

  //must be called by OS
  if (core_info_enclave_id ≠null_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  if (level < 0 ∨level > 2) {
    result :=monitor_invalid_value;
    return;
  }

  if (¬is_dram_address(paddr)) {
    result :=monitor_invalid_value; return;
  }

  if (¬is_page_aligned_pa(paddr)) {
    result :=monitor_invalid_value; return;
  }

  // valid enclave?
  if (eid ≠null_enclave_id ∧¬is_valid_enclave_id(enclave_metadata_valid, eid)) {
    result :=monitor_invalid_value;
    return;
  }

  if (eid =null_enclave_id ∧level ≠2) {
    result :=monitor_invalid_value;
    return;
  }

  if (enclave_metadata_is_initialized[eid]) {
    result :=monitor_invalid_state;
    return;
  }
```

```
  // FIXME: verify this.
  // we don't own this region so can't allow page tables to be outside it.
  paddr_region :=dram_region_for(paddr);
  if (owner[paddr_region] ≠eid ∧level > 0) {
    result :=monitor_invalid_value;
    return;
  }

  if (level ≠2 ∧
      ((eid =null_enclave_id ∧cpu_ptbr =k0_ppn_t) ∨
       (eid ≠null_enclave_id ∧enclave_metadata_load_eptbr[eid] =k0_ppn_t)))
  {
    result :=monitor_invalid_state;
    return;
  }

  if (level =2) {
    if (eid =null_enclave_id) {
      cpu_ptbr :=paddr2ppn(paddr);
    } else {
      enclave_metadata_load_eptbr[eid] :=paddr2ppn(paddr);
      call _clear_mapped_pages(paddr2ppn(paddr));
      assert (∀ p : ppn_t, v : vpn_t •if p =paddr2ppn(paddr)
                 then ptbl_acl_map[p, v] =k_pg_invalid_acl
                 else ptbl_acl_map[p, v] =old(ptbl_acl_map)[p, v]);
      assert (∀ e: enclave_id_t •e ≠eid ⟹enclave_metadata_load_eptbr[e] =old(enclave_metadata_load_eptbr[e]));
    }
    result :=monitor_ok;
  } else {
    ptbr :=if (eid =null_enclave_id) then cpu_ptbr else enclave_metadata_load_eptbr[eid];
    // update page tables.
    call success :=create_page_table_mapping(ptbr, vaddr, paddr, acl);
    result :=if (success) then monitor_ok else monitor_unknown_error;
    // update measurement.
    measurement :=enclave_metadata_measurement[eid];
    measurement :=update_digest(measurement, code_api_load_page_table);
    measurement :=update_digest(pte_acl2int(acl), measurement);
    measurement :=update_digest(vaddr2int(vaddr), measurement);
    enclave_metadata_measurement[eid] :=measurement;

    assert (result =monitor_ok ⟹ptbl_acl_map[enclave_metadata_load_eptbr[eid], vaddr2vpn(vaddr)] =acl);
    assert (result =monitor_ok ⟹translate_vaddr2paddr(ptbl_addr_map, enclave_metadata_load_eptbr[eid], vaddr) =
                                 (paddr2ppn(paddr) ++ vaddr2offset(vaddr)));
  }
}

//api_result_t load_page(enclave_id_t enclave_id, uintptr_t phys_addr, uintptr_t virtual_addr, uintptr_t os_addr, uintptr_t acl);
procedure load_page_impl(eid: enclave_id_t, vaddr: vaddr_t, src_paddr : paddr_t)
  returns (result: api_result_t);
  modifies mem;
  ensures ((core_info_enclave_id ≠null_enclave_id)                                    ∨
           (¬is_page_aligned_va(vaddr))                                               ∨
           (¬is_dram_address(src_paddr))                                              ∨
           (¬is_page_aligned_pa(src_paddr))                                           ∨
           (eid =null_enclave_id ∨¬is_valid_enclave_id(enclave_metadata_valid, eid))  ∨
           (enclave_metadata_is_initialized[eid])                                     ∨
           (¬is_in_evrange(enclave_metadata_ev_base[eid], enclave_metadata_ev_mask[eid], vaddr)) ∨
           (enclave_metadata_load_eptbr[eid] =k0_ppn_t)                               ∨
           (¬acl2valid(ptbl_acl_map[enclave_metadata_load_eptbr[eid], vaddr2vpn(vaddr)]))) ⟺
          (result ≠monitor_ok);

  ensures (result ≠monitor_ok) ⟹(mem =old(mem));
  ensures (result =monitor_ok) ⟹
           (∀ p : wap_addr_t •
             if (wpaddr2ppn(p) =ptbl_addr_map[enclave_metadata_load_eptbr[eid], vaddr2vpn(vaddr)])
                 then (mem[p] =mem[paddr2ppn(src_paddr) ++ wpaddr2offset(p)])
                 else (mem[p] =old(mem)[p]));

procedure {: inline 1} load_page(eid: enclave_id_t, vaddr: vaddr_t, src_paddr : paddr_t)
  returns (result: api_result_t)
  modifies mem;
  modifies enclave_metadata_measurement;
  ensures ((core_info_enclave_id ≠null_enclave_id)                                    ∨
           (¬is_page_aligned_va(vaddr))                                               ∨
           (¬is_dram_address(src_paddr))                                              ∨
           (¬is_page_aligned_pa(src_paddr))                                           ∨
           (eid =null_enclave_id ∨¬is_valid_enclave_id(enclave_metadata_valid, eid))  ∨
           (enclave_metadata_is_initialized[eid])                                     ∨
           (¬is_in_evrange(enclave_metadata_ev_base[eid], enclave_metadata_ev_mask[eid], vaddr)) ∨
           (enclave_metadata_load_eptbr[eid] =k0_ppn_t)                               ∨
           (¬acl2valid(ptbl_acl_map[enclave_metadata_load_eptbr[eid], vaddr2vpn(vaddr)]))) ⟺
          (result ≠monitor_ok);

  ensures (result ≠monitor_ok) ⟹(mem =old(mem));
  ensures (result =monitor_ok) ⟹
```

```
          (∀ p : wap_addr_t •
            if (wpaddr2ppn(p) =ptbl_addr_map[enclave_metadata_load_eptbr[eid], vaddr2vpn(vaddr)])
                then (mem[p] =mem[paddr2ppn(src_paddr) ++ wpaddr2offset(p)])
                else (mem[p] =old(mem)[p]));
{
    var m : measurement_t;
    call result :=load_page_impl(eid, vaddr, src_paddr);
    if (result =monitor_ok) {
        m :=enclave_metadata_measurement[eid];
        m :=update_digest(m, code_api_load_page);
        m :=update_digest(m, vaddr2int(vaddr));
        call m :=measure_mem(m, paddr2wpaddr(src_paddr), mem, kPGSZ_wap_addr_t);
        //enclave_metadata_measurement[eid] :=m;
    }
}

procedure {: inline 1} assign_dram_region(region: region_t, new_owner: enclave_id_t)
  returns (result: api_result_t)
  modifies owner, enclave_metadata_bitmap, os_bitmap, cpu_drbmap;
{
  //must be called by OS
  if (core_info_enclave_id ≠null_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  if (¬is_valid_dram_region(region)) {
    result :=monitor_invalid_value; return;
  }

  if (¬enclave_metadata_valid[new_owner]) {
    result :=monitor_invalid_value; return;
  }

  if (enclave_metadata_is_initialized[new_owner]) {
    result :=monitor_invalid_state; return;
  }

  //can only assign free dram regions
  if (owner[region] ≠free_enclave_id) {
    result :=monitor_invalid_state; return;
  }

  owner[region] :=new_owner;
  if (new_owner =null_enclave_id) {
    os_bitmap :=bitmap_set_bit(os_bitmap, region);
    cpu_drbmap :=os_bitmap;
  } else {
    enclave_metadata_bitmap[new_owner] :=bitmap_set_bit(enclave_metadata_bitmap[new_owner], region);
  }

  result :=monitor_ok;
}

procedure {: inline 1} load_thread(eid: enclave_id_t, tid: thread_id_t, entry_pc: vaddr_t, entry_stack: vaddr_t, fault_pc: vaddr_t,
      fault_stack: vaddr_t)
  returns (result: api_result_t)
  modifies thread_metadata_valid,
           thread_metadata_eid,
           thread_metadata_entry_pc,
           thread_metadata_entry_stack,
           thread_metadata_fault_pc,
           thread_metadata_fault_stack,
           enclave_metadata_thread_count,
           enclave_metadata_measurement;
{
  var measurement : measurement_t;

  // must be called by OS
  if (core_info_enclave_id ≠null_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  if (¬is_valid_enclave_id(enclave_metadata_valid, eid)) {
    result :=monitor_invalid_value; return;
  }

  if (enclave_metadata_is_initialized[eid]) {
    result :=monitor_invalid_state; return;
  }
  //if (enclave_metadata_load_eptbr[eid] =k0_paddr_t) {
  //  result :=monitor_invalid_state; return;
  //}
  // tid's are integers and not pointers.
  //if (tid ≠enclave_metadata_thread_count[eid]) {
  //  result :=monitor_invalid_state; return;
  //}
```

```
  // set thread metadata.
  thread_metadata_valid[tid] :=true;
  thread_metadata_eid[tid] :=eid;
  thread_metadata_entry_pc[tid] :=entry_pc;
  thread_metadata_entry_stack[tid] :=entry_stack;
  thread_metadata_fault_pc[tid] :=fault_pc;
  thread_metadata_fault_stack[tid] :=fault_stack;
  // enclave metadata.
  enclave_metadata_thread_count[eid] :=enclave_metadata_thread_count[eid] + 1;
  // update measurement.
  measurement :=enclave_metadata_measurement[eid];
  measurement :=update_digest(code_api_load_thread, measurement);
  measurement :=update_digest(vaddr2int(entry_pc), measurement);
  measurement :=update_digest(vaddr2int(entry_stack), measurement);
  measurement :=update_digest(vaddr2int(fault_pc), measurement);
  measurement :=update_digest(vaddr2int(fault_stack), measurement);
  measurement :=update_digest(enclave_metadata_thread_count[eid], measurement);
  enclave_metadata_measurement[eid] :=measurement;

  result :=monitor_ok;
}

procedure {: inline 1} init_enclave(eid: enclave_id_t)
  returns (result: api_result_t)
  modifies enclave_metadata_is_initialized;
{
  //must be called by OS
  if (core_info_enclave_id ≠null_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  if (¬assigned(eid) ∨¬is_valid_enclave_id(enclave_metadata_valid, eid)) {
    result :=monitor_invalid_state; return;
  }
  if (enclave_metadata_is_initialized[eid]) {
    result :=monitor_invalid_state; return;
  }
  if (owner[dram_region_for(enclave_metadata_load_eptbr[eid] ++ 0bv12)] ≠eid) {
    result :=monitor_invalid_state; return;
  }

  enclave_metadata_is_initialized[eid] :=true;
  result :=monitor_ok;
}


procedure {: inline 1} enter_enclave(eid: enclave_id_t, tid: thread_id_t)
  returns (result: api_result_t)
  modifies core_info_enclave_id,
           core_info_thread_id,
           cpu_evbase,
           cpu_evmask,
           cpu_edrbmap,
           cpu_eptbr,
           rip;
{
  //must be called by OS
  if (core_info_enclave_id ≠null_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  if (¬assigned(eid) ∨¬is_valid_enclave_id(enclave_metadata_valid, eid)) {
    result :=monitor_invalid_value; return;
  }
  if (¬enclave_metadata_valid[eid] ∨
      ¬thread_metadata_valid[tid] ∨
      thread_metadata_eid[tid] ≠eid)
  {
    result :=monitor_invalid_state; return;
  }
  if (¬enclave_metadata_is_initialized[eid]) {
    result :=monitor_invalid_state; return;
  }

  core_info_enclave_id :=eid;
  core_info_thread_id :=tid;
  cpu_evbase :=enclave_metadata_ev_base[eid];
  cpu_evmask :=enclave_metadata_ev_mask[eid];
  cpu_edrbmap :=enclave_metadata_bitmap[eid];
  cpu_eptbr :=enclave_metadata_load_eptbr[eid];
  // FIXME: cpu_ptbr? What happens to this?

  rip :=thread_metadata_entry_pc[tid];

  result :=monitor_ok;
```

```
}

procedure {: inline 1} exit_enclave()
  returns (result: api_result_t)
  modifies core_info_enclave_id, rip;
{
  //must be called by OS
  if (core_info_enclave_id ==null_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  core_info_enclave_id :=null_enclave_id;
  rip :=os_pc;
  result :=monitor_ok;
}

procedure {: inline 1} block_dram_region(region: region_t)
  returns (result: api_result_t)
  modifies owner,
           enclave_metadata_bitmap,
           os_bitmap, cpu_drbmap,
           cpu_edrbmap, blocked_at,
           dram_regions_info_block_clock;
{
  if (¬is_dynamic_dram_region(region)) {
    result :=monitor_invalid_value; return;
  }

  if (owner[region] ≠core_info_enclave_id) {
    result :=monitor_access_denied; return;
  }

  if (owner[region] ==null_enclave_id) {
    os_bitmap :=bitmap_clear_bit(os_bitmap, region);
    cpu_drbmap :=os_bitmap;
  } else {
    enclave_metadata_bitmap[owner[region]] :=bitmap_clear_bit(enclave_metadata_bitmap[owner[region]], region);
    cpu_edrbmap :=enclave_metadata_bitmap[owner[region]];
  }

  owner[region] :=blocked_enclave_id;
  blocked_at[region] :=dram_regions_info_block_clock;
  dram_regions_info_block_clock :=dram_regions_info_block_clock + 1;

  result :=monitor_ok;
}

// Frees a DRAM region that was previously blocked.
procedure free_dram_region(region: region_t)
  returns (result: api_result_t);
  modifies owner, mem; //, os_bitmap;
  ensures (result ==monitor_ok ∨result ==monitor_invalid_value ∨result ==monitor_invalid_state);
  ensures (is_valid_dram_region(region) ∧old(owner)[region] ==blocked_enclave_id ∧core_flushed_at ≥blocked_at[region])
         ⟺(result ==monitor_ok);
  ensures (result ≠monitor_ok) ⟹(owner ==old(owner));
  ensures (result ≠monitor_ok) ⟹(mem ==old(mem));
  //ensures (result ≠monitor_ok) ⟹(os_bitmap ==old(os_bitmap));
  ensures (result ==monitor_ok) ⟹
         (∀ r : region_t •if r ==region then owner[r] ==free_enclave_id
                                        else owner[r] ==old(owner)[r]);
  ensures (result ==monitor_ok) ⟹
         (∀ p : wap_addr_t •if dram_region_for_w(p) ==region
                           then mem[p] ==k0_word_t
                           else mem[p] ==old(mem)[p]);
  //ensures (result ==monitor_ok) ⟹
  //       (os_bitmap ==bitmap_set_bit(os_bitmap, region));

procedure {: inline 1} flush_cached_dram_regions()
  returns (result: api_result_t)
  modifies core_flushed_at;
{
  //must be called by OS
  if (core_info_enclave_id ≠null_enclave_id) {
    result :=monitor_invalid_value; return;
  }

  //hardware TLB flush, which we won't model
  core_flushed_at :=dram_regions_info_block_clock;
  result :=monitor_ok;
}

procedure {: inline 1} delete_thread(tid: thread_id_t)
  returns (result: api_result_t)
  modifies thread_metadata_valid, enclave_metadata_thread_count;
{
  var eid: enclave_id_t;
```

```
  // must be called by OS
  if (core_info_enclave_id ≠null_enclave_id) {
    result :=monitor_invalid_value; return;
  }
  // is this a valid thread?
  if (¬thread_metadata_valid[tid]) {
    result :=monitor_invalid_value; return;
  }
  // find enclave id.
  eid :=thread_metadata_eid[tid];
  // (for now), can only be called before initialization.
  if (enclave_metadata_is_initialized[eid]) {
    result :=monitor_invalid_state; return;
  }

  thread_metadata_valid[tid] :=false;
  enclave_metadata_thread_count[eid] :=enclave_metadata_thread_count[eid] - 1;
  result :=monitor_ok;
}

procedure delete_enclave(eid: enclave_id_t)
  returns (result: api_result_t);
  modifies enclave_metadata_valid, enclave_metadata_is_initialized, owner;

  // result is one of these values.
  ensures (result =monitor_ok ∨result =monitor_invalid_value ∨result =monitor_invalid_state);
  // conditions for success.
  ensures (core_info_enclave_id =null_enclave_id ∧
          assigned(eid)                          ∧
          old(enclave_metadata_valid)[eid]       ∧
          enclave_metadata_thread_count[eid] =0) ⟺
         (result =monitor_ok);
  // no changes upon failure.
  ensures (result ≠monitor_ok) ⟹
           (enclave_metadata_valid =old(enclave_metadata_valid)            ∧
            enclave_metadata_is_initialized =old(enclave_metadata_is_initialized) ∧
            owner =old(owner));
  // must destroy enclave metadata on success.
  ensures (result =monitor_ok) ⟹¬enclave_metadata_valid[eid];
  ensures (result =monitor_ok) ⟹¬enclave_metadata_is_initialized[eid];
  ensures (result =monitor_ok) ⟹(∀ ri : region_t •owner[ri] ≠eid);
  ensures (result =monitor_ok) ⟹
        (∀ ri : region_t •
           if old(owner)[ri] =eid
              then owner[ri] =blocked_enclave_id
              else owner[ri] =old(owner)[ri]);
  // change only enclave metadata for e.
  ensures (∀ e : enclave_id_t •e ≠eid ⟹
                  (enclave_metadata_valid[e] =old(enclave_metadata_valid[e])));
  ensures (∀ e : enclave_id_t •e ≠eid ⟹
                  (enclave_metadata_is_initialized[e] =old(enclave_metadata_is_initialized[e])));
```

# Appendix D

# Proof of Theorem 3 and Theorem 4

## D.1 Preliminaries

While we have a machine-checked proof of Theorem 3 and Theorem 4 (stated in Chapter 8), we describe our proof skeleton in this chapter. We prove both Theorem 3 and 4 by performing automated verification on a meta-program, which is specified using BoogiePL syntax [14]. Since the language used to specify $U_M$ (defined in Figure 3) is not BoogiePL, we first define the BoogiePL equivalent of each statement in our language.

- reg := load(a) is syntactic sugar for reg := mem[a].
- store(a, d) is syntactic sugar for mem := mem[a := d].
- `jmp` $l$ is syntactic sugar for rip := $l$, where rip is the instruction pointer
- `call` $l$ is syntactic sugar for rsp := rsp − 8; mem[rsp] := next(rip); rip := $l$, where next returns the next x64 instruction.
- ret is syntactic sugar for rip := mem[rsp]; rsp := rsp + 8.
- havoc $\phi$ statement scrambles all memory locations specified by predicate $\phi$. Intuitively, this creates a new memory $\texttt{mem}'$ whose content is the same as in memory `mem` for all addresses that do not satisfy $\phi$, as follows:

$$\texttt{assume } \forall a.\ \neg\phi(a) \Rightarrow \texttt{mem}'[a] = \texttt{mem}[a];$$
$$\texttt{mem} := \texttt{mem}'$$

`assert`, `assume` and skip statements are present in BoogiePL, and are semantically equivalent.

## D.2 Proof of Theorem 3

We prove Theorem 3 by performing automated verification on a meta-program, which we describe below. The meta-program models all possible user programs $U_M$ using non-deterministic choice of instruction sequences, and asserts the condition needed to prove Theorem 3. We prove the assertion of the meta-program using the Boogie program verifier [14], thus proving Theorem 3.

We describe the structure of the meta-program $p_1$ in Figure D.1 (written in Boogie syntax). Program $p_1$ models an arbitrary execution of $U_M$ i.e. each trace of $U_M$ is an allowed trace of $p_1$. To model unbounded length executions containing arbitrary statements, $p_1$ has a while loop (line 10) that arbitrarily chooses a statement from the set $\{\texttt{call}\ l, \mathsf{ret}, \texttt{jmp}\ l, \mathsf{reg} := a, \mathsf{store}(\mathrm{a}, \mathrm{d}), \mathsf{reg} := \mathsf{load}(\mathrm{a}), \mathsf{havoc}\ \texttt{mem}_{\neg\texttt{evrange}}$ and executes it with symbolic values for arguments of the statement (e.g. a in $\mathsf{reg} := \mathsf{load}(\mathrm{a})$). The symbolic values are unconstrained—we havoc them in each iteration (line 13). A `call` may either target a procedure in $U_M$ (line 31) or an API of $L$ (lines 37, 39, 41, 44). In the case of call to a procedure in $U_M$, we transfer control to the new procedure and continue execution (as a processor would do). In the case of call to $L$'s API, the call is modeled by havocing memory and registers and assuming the postconditions of the API (because the postconditions of $L$ are proved correct separately).

The intuition behind the proof is as follows. Theorem 3 states that if $U_M$ satisfies WCFI-RW and $L$ satisfies the functional correctness properties (given in Section 8.2.2), then $U_M$ can only modify non-enclave memory by invoking the send API of $L$ or by having $M$ invoke $\mathsf{havoc}\ \texttt{mem}_{\neg\texttt{evrange}}$. In order to make an arbitrary $U_M$ satisfy WCFI-RW, we effectively compose $U_M$ with the monitor automaton $\mathcal{W}$ (defined in Definition 6). That is, each statement of $U_M$ must also satisfy the validity check on the corresponding transition in $\mathcal{W}$. This is encoded as `assume` statements in $p_1$ (lines 19, 22, 25, 28, 31, 34, 37, 39, 41, 44). To prove this theorem, we prove that for any statement $i \in \texttt{instr}(\pi)$, if the validity checks in $\mathcal{W}$ are valid, then executing that statement preserves IRC.

We prove Theorem 3 inductively by asserting the IRC property after each statement of $U_M$ in line 51. After manually supplying the loop invariant $\mathsf{legal}(\mathsf{rip})$ && $\mathsf{AddrInU}(\mathsf{rsp})$, Boogie is able to verify this program. The loop invariant guarantees that the input received by $\mathcal{W}$ corresponds to the instruction executed by the processor, making our meta program $p_1$ sound. Temporal induction on the length of $U_M$'s execution establishes that either an `assert` in $p_1$ will fail, or $p_1$ will over-approximate an arbitrary $U_M$.

## D.3 Proof of Theorem 4

We prove Theorem 4 by performing automated verification on a meta-program, which we describe below. The meta-program models all possible user programs $U_M$ using non-deterministic choice of instruction sequences, and asserts the conditions needed to prove Theorem 4. We prove the assertions of the meta-program using the Boogie program veri-

fier [14], thus proving the theorem.

The proof of Theorem 4 uses induction over the length of an execution. Theorem 4 states that any execution $\pi$ (of unbounded length) that satisfies each assertion in $I(\texttt{instr}(\pi))$ must satisfy WCFI-RW. To prove this theorem, we prove that for any statement $i \in \texttt{instr}(\pi)$, if the assertions in $I(i)$ are valid, then there exists a valid transition in $\mathcal{W}$ on input $i$. Our meta-program models this by assuming that the assertions in $I(i)$ are valid, and then asserting the validity check on the corresponding transition in $\mathcal{W}$. To that end, we introduce $\hat{I} : \mathit{Instr} \rightarrow \mathit{Instr}$, which replaces each occurrence of `assert` in the definition of $I$ with `assume`.

We describe the structure of the meta-program $p_2$ in Figure D.2 (written in Boogie syntax). Program $p_2$ models the execution of an arbitrary procedure of $U_M$. The execution of a procedure starts with a `call` statement (line 8) and ends in a ret statement (line 34), with an unbounded number of statements in between. To model unbounded length executions containing arbitrary statements, $p_2$ has a while loop (line 10) that arbitrarily chooses a statement from the set $\{\texttt{call}\ l, \mathsf{ret}, \texttt{jmp}\ l, \mathsf{reg} := e, \mathsf{store}(\mathrm{a}, \mathrm{d}), \mathsf{reg} := \mathsf{load}(\mathrm{a})\}$ and executes it with symbolic values for arguments of the statement (e.g. a in reg := load(a)). The symbolic values are unconstrained—we havoc them in each iteration (line 19). Calling a procedure of $U_M$ is modeled by recursively calling `p` itself (line 31). This recursive call allows $p_2$ to encode executions of $U_M$ that reach arbitrary stack depths. We also allow $p_2$ to call APIs of $L$, but this can also be modeled as a recursive call to `p` — we additionally check that the functional contracts on $L$ imply the postcondition $\phi$ (lines 37, 38, 39, 40) Note that we also assume that $U_M$ cannot read from $L$'s memory (line 25), which is automatically enforced by the CPU based on the configuration of the page tables.

The justification of why $p_2$ models an arbitrary $U_M$ involves an inductive use of WCFI-RW itself. The argument requires that: (1) $U_M$ cannot jump into the code of L at locations other than the entrypoints of L and (2) $U_M$ follows a stack discipline of procedure calls and returns. Each of these properties are assumed for the construction of $p_2$ but also asserted by $p_2$ itself. For each control transfer statement in $U_M$ (`call` and `jmp` ) the policy constrains the target location. Further, the assertions on ret enforce the stack discipline. Temporal induction on the length of U's execution establishes that that either an assertion in $p_2$ will fail, or it will over-approximate an arbitrary $U_M$.

We manually supply an inductive invariant (lines 11, 14, 17), after which Boogie is able to proves all assertions in $p_2$, thereby proving Theorem 4.

```
1  var mem: [bv64] bv8,
2  var rax, rsp, ...: bv64;
3  var rip: bv64;
4  procedure U();
5    modifies mem, rax, rsp, rip, ... ;
6  {
7    var addr, data, target: bv64;
8    var buf, size: bv64;
9
10   while (*)
11   invariant legal(rip) && AddrInU(rsp);
12   {
13     havoc a, d, l;
14     mem_old := mem;
15     is_send := false;
16     is_havoc := false;
17
18     if (*) {
19         assume ψrsp
20         rsp := a;
21     } else if (*) {
22         assume ψload;
23         rax/... := load(a);
24     } else if (*) {
25         assume ψstore;
26         store(a,d);
27     } else if (*) {
28         assume ψjmp;
29         jmp l;
30     } else if (*) {
31         assume ψcall;
32         call l;
33     } else if (*) {
34         assume ψret;
35         ret;
36     } else if (*) {
37         havoc mem,rax,...; assume ψmalloc;
38     } else if (*) {
39         havoc mem,rax,...; assume ψfree;
40     } else if (*) {
41         havoc mem,rax,...; assume ψsend;
42         is_send := true;
43     } else if (*) {
44         havoc mem,rax,...; assume ψrecv;
45     } else if (*) {
46       havoc mem;
47       assume ∀i. ¬evrange(i)⇒ mem[i] = mem_old[i];
48         is_havoc := true;
49     }
50
51     assert (¬is_send && ¬is_havoc) ⇒
52         (∀i. ¬evrange(i) ⇒ mem[i] = mem_old[i]);
53   }
54 }
```

*Figure D.1*: Meta Program $p_1$

```
1  var mem: [bv64] bv8, rax: bv64, ...;
2  procedure p() ensures φ;
3    modifies mem, rax...;
4  {
5    var a, d, l;
6    var old_rsp: bv64;
7    var mem_old : [bv64] bv8;
8    Î(call p); assert ψ_call;
9    old_rsp := rsp; mem_old := mem;
10   while (*)
11   invariant
12     ∀i. (AddrInStack(i) && i >= old_rsp) ⇒
13       (writable(mem,i) = writable(mem_old,i));
14   invariant
15     ∀i (AddrInStack(i) && i >= old_rsp + 40 && ¬writable(mem,i)) ⇒
16       (mem[i] = old_mem[i]);
17   invariant rsp <= old_rsp && rsp[3:0] = 000;
18   {
19     havoc a, d, l;
20     mem_old := mem;   b_send := false;
21     if (*) {
22       Î(rsp := d); assert ψ_rsp;
23     } else if (*) {
24       Î(rax/... := load(a));
25       assume ¬AddrInL(a); assert ψ_load;
26     } else if (*) {
27       Î(store(a,d)); assert ψ_store;
28     } else if (*) {
29       Î(jmp l); assert ψ_jmp;
30     } else if (*) {
31       havoc mem, rax,...; assume φ
32     }
33   }
34   Î(ret); assert ψ_ret;
35 }
36
37 assert ψ_malloc ⇒ φ;
38 assert ψ_free ⇒ φ;
39 assert ψ_send ⇒ φ;
40 assert ψ_recv ⇒ φ;
```

*Figure D.2*: Meta Program $p_2$