

LuaGuardia: A Confidential Computing Framework for Trusted Execution Environments

Dimitrios Karnikis

Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisors: Prof. Polyvios Pratikakis, Prof. Sotiris Ioannidis

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

This work was partially supported by **Institute of Computer Science, Foundation of Research and Technology Hellas**

This work was partially supported by **Institute of Computer Science, Foundation of Research and Technology Hellas**

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

**LuaGuardia: A Confidential Computing Framework for Trusted Execution
Environments**

Thesis submitted by
Dimitrios Karnikis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

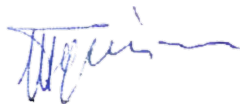
THESIS APPROVAL

Author:

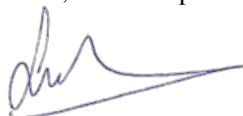


Dimitrios Karnikis

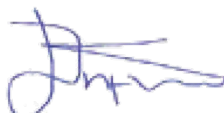
Committee approvals:



Polyvios Pratikakis
Professor, Thesis Supervisor, Committee Member

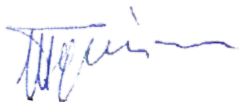


Sotiris Ioannidis
Associate Professor, Thesis Advisor, Committee Member



Xenofontas Dimitropoulos
Professor, Committee Member

Departmental approval:



Polyvios Pratikakis
Professor, Director of Graduate Studies

Heraklion, January 2021

Abstract

Confidential computing applications are enabled by Trusted Execution environments (TEEs) that are becoming increasingly widespread in the computing landscape. However, their development and deployment remains challenging due to several reasons. The lack of high-level TEE abstractions complicates application development and forces the use of low-level memory- and type-unsafe abstractions. These challenges are exacerbated by technical issues regarding runtime extensibility, management of cryptographic operations, and restricted interfaces: even porting existing applications requires manual partitioning, re-compilation, and linking steps.

This work presents LuaGuardia, a system simplifying the development of confidential computing. LuaGuardia addresses the aforementioned challenges by offering a set of abstractions around a TEE-embedded runtime environment of a high-level programming language. LuaGuardia’s abstractions simplify the development and deployment of such applications in a type- and memory-safe manner. It also offers a runtime library solving technical challenges such as code signing, system-call offloading, access control, and dynamic code loading. A series of optimizations is also provided that accelerate protected code execution. Our evaluation applies LuaGuardia to a diverse set of applications, cryptographic functions but also real-world commercial applications, with an average overhead of 18%, the majority of which is due to I/O delays.

Περίληψη

Οι εφαρμογές εκτελούν κώδικα κρίσιμο για την ασφάλεια ενός συστήματος και διαχειρίζονται ευαίσθητα ή προσωπικά δεδομένα. Συχνά υποστηρίζονται από Περιβάλλοντα Ασφαλούς Εκτέλεσης (ΠΑΕ), τα οποία γίνονται όλο και πιο διαδεδομένα στον χώρο των υπολογιστών. Ωστόσο, η ανάπτυξη και η εφαρμογή τους παραμένουν μία πρόκληση για διάφορους λόγους. Αρχικά, η έλλειψη διεπαφών υψηλού επιπέδου στα ΠΑΕ περιπλέκει την ανάπτυξη εφαρμογών και αναγκάζει τη χρήση γλωσσών που δε προσφέρουν ασφαλείς διεπαφές μνήμης και τύπων. Αυτές οι προκλήσεις επιδεινώνονται από τεχνικά ζητήματα που αφορούν την επεκτασιμότητα του περιβάλλοντος εκτέλεσης, την διαχείριση των κρυπτογραφικών λειτουργιών αλλά και τις περιορισμένες διαθέσιμες διεπαφές. Επίσης, η μεταφορά των υπάρχοντων εφαρμογών σε διαφορετικά συστήματα/πλατφόρμες απαιτεί χειροκίνητη τμηματοποίηση της εφαρμογής, εκ νέου μεταγλώττιση του πηγαίου κώδικα και την υλοποίηση των απαραίτητων βημάτων σύνδεσης για την τελική δημιουργία του εκτελέσιμου προγράμματος.

Αυτή η εργασία παρουσιάζει το **LuaGuardia**, ένα σύστημα που απλοποιεί την ανάπτυξη εφαρμογών βασισμένων σε ΠΑΕ. Το **LuaGuardia** αντιμετωπίζει τις προαναφερθείσες προκλήσεις προσφέροντας ένα σύνολο διεπαφών υψηλού επιπέδου γύρω από ένα περιβάλλον εκτέλεσης σε γλώσσα υψηλού επιπέδου. Αυτές οι διεπαφές απλοποιούν την ανάπτυξη τέτοιων προγραμμάτων με γνώμονα τη διατήρηση της ασφάλειας των τύπων και την προστασία της μνήμης. Ακόμη, προσφέρει μια βιβλιοθήκη που επιλύει τεχνικές δυσκολίες όπως την υπογραφή του εκτελέσιμου κώδικα, τη διαχείριση των κλήσεων συστήματος, τον έλεγχο πρόσβασης σε δεδομένα και τη δυναμική φόρτωση κώδικα. Ακόμη, προσφέρει μια σειρά βελτιστοποιήσεων που επιταχύνουν την εκτέλεση προστατευμένου κώδικα. Επιπλέον, σε αυτή την εργασία αξιολογούμε την επίδοση του **LuaGuardia** σε ένα σύνολο από αλγορίθμους, κρυπτογραφικές συναρτήσεις αλλά και εμπορικές εφαρμογές. Μέσω πειραμάτων παρατηρούμε ότι το σύστημά μας προσφέρει τα παραπάνω πλεονεκτήματα, αυξάνοντας κατά μέσο όρο τον χρόνο εκτέλεσης κατά 18%, με την πλειονότητα των χρονικών επιβαρύνσεων να οφείλεται σε καθυστερήσεις εισόδου/εξόδου.

Acknowledgments

First of all, I would like to thank my supervisor, Professor Polyvios Pratikakis, for his valuable guidance and all the constructive conversations we had. I also want to express my deepest gratitude to my advisor, Professor. Sotiris Ioannidis, for giving me the opportunity to work on so many different, challenging and interesting projects, over the past three years. His support and advice greatly contributed to my academic and technical growth. Moreover, I feel thankful to my colleague, mentor and friend Dimitri Deyanni, for his guidance during my first steps of my academic journey from bachelor years till my masters.

My warmest regards to all the members of the DiSCs Laboratory, for their friendship, advice and commitment.

Finally, I want to thank my family and friends for bearing with me and for providing their invaluable support and caring through all these years.

Contents

1	Introduction	1
1.1	Guarantees	2
1.2	Outline	3
2	Background	5
2.1	Trusted Execution Environments	5
2.1.1	Intel Software Guard Extensions	6
2.1.2	ARM TrustZone	9
2.2	Just-In-Time Languages	10
2.2.1	Why Lua	10
2.3	Motivating Examples	11
2.3.1	LuaGuardia Overview	12
3	Threat Model and Assumptions	15
4	System Architecture	17
4.1	LVMAT Components	17
4.1.1	LVMAT Interpreter	17
4.1.2	LVMAT Server	18
4.1.3	LVMAT Client Stub	19
4.2	LuaGuardia Local Execution Mode	19
5	System Implementation	21
5.1	Porting the Lua VM	21
5.2	system-call Handling	22
5.3	External Modules	24
5.4	Maintaining Global State	25
5.5	Code and Client Isolation	25
5.6	Optimizations	27
6	System Evaluation	29
6.1	Experimental Testbed	29
6.1.1	Security Analysis	29
6.2	Micro Benchmarks	31

6.2.1	Benchmark Applications	33
6.2.2	Performance Optimizations	34
6.2.3	Real World Application 1: wrk2	36
6.2.4	Real World Application 2: Snabb/pflua	38
6.2.5	Real World Application 3: Snabb/IPsec	41
7	Discussion And Limitations	43
7.1	Local Execution Mode	43
7.2	Native Module Support	43
7.3	Enclave Size	44
8	Related Work	45
9	Conclusions and Future Work	47
9.1	Summary of Contributions	47
9.2	Future Work	47
9.3	Conclusion	48

List of Tables

5.1	Most used functions supported by LuaGuardia	24
6.1	Benchmark Operations	35

List of Figures

2.1	A normal trusted function execution. The untrusted requests the creation of the enclave and then invokes the requested function. This request is forwarded to the Trusted bridge that performs certain checks and validations, and then passes the request with its arguments to the respective function inside the enclave. The code inside the trusted part is executed inside the EPC cache and the results are fetched back to the call site. The code then resumes its normal execution.	8
2.2	Exception Levels and Security states on Armv8-a architecture.	10
2.3	LuaGuardia overview	13
5.1	Potential risks on running a simple Lua script on the untrusted environment. In LuaGuardia, all the untrusted I/O is handled via encryption, libraries and data are stored encrypted in the untrusted file-system and the attacker may not extract any useful information by observing the enclave code.	28
6.1	Performance comparison between the vanilla Lua virtual machine and LuaGuardia when reading a 32MB file with variable read buffer sizes	31
6.2	Performance comparison between the vanilla Lua virtual machine and LuaGuardia when randomly performing one million accesses (read/write) to memory locations ranging between 1KB and 4MB	32
6.3	Performance sustained for cryptographic benchmarks	34
6.4	End-to-end performance when executing 12 popular Lua benchmarks.	36
6.5	LuaGuardia execution time breakdown with and without server initialization optimizations.	37
6.6	Performance comparison between the vanilla Lua virtual machine, LuaGuardia and LuaGuardia with system-call batching enabled.	38
6.7	Performance sustained when applying system-call batching and enclave pre-initialization.	39
6.8	Performance sustained for wrk2	40
6.9	Performance sustained for pflua	41
6.10	Performance sustained for IPsec	42

Chapter 1

Introduction

Confidential computing protects data while in use while they are in use, by isolating computations using a hardware-based trusted execution environment(TEE). The main and primary goal of a TEE is to isolate programs, program fragments, and their data from potentially malicious operating systems, hypervisors, and other privileged or not processes. TEEs that are becoming increasingly widespread in the computing landscape [1, 2, 3], enabling confidential computing applications in analytics, medicine, telemetry and several other domains.

Unfortunately, developing such confidential computing applications is far from trivial due to several challenges [4, 5, 6, 7]. This occurs due to TEE lacking high-level abstractions, a challenge that (i) complicate application development for users that have no prior TEE knowledge (ii) forces developers to use low-level memory-and type-unsafe abstractions. As a result, the lack of type and memory safety affects the security of confidential computing applications which are the applications that TEEs were developed to support in the first place. Additionally, moving even the critical part of the application requires a lot of manual work such as code partitioning, re-compilation and linking the entire application including the components of the application that run outside of the trusted environment. Such static code requirements impair the development and use of certain applications that require code extensibility.

In this work, we present LuaGuardia, a system aimed at simplifying the development of confidential computing on TEEs. LuaGuardia offers a set of high-level abstractions for building applications from the scratch or porting existing legacy applications to the TEE. LuaGuardia’s key insight is to embed the runtime environment of a high-level programming language into the TEE. In LuaGuardia’s case, the language is Lua and the TEE is based on Intel SGX. We opted for Lua due to its high-level semantics, its dynamic meta-programming facilities(which include runtime code evaluation), and its small memory footprint: the original Lua interpreter consists of 19K Lines of Code(LoC), consuming minimal enclave memory thus leaving more memory resources available for applications that are built atop of LuaGuardia. LuaGuardia packages a small runtime library for dealing with application signing, extensibility, system-call forwarding and other technical challenges.

To test the effectiveness of LuaGuardia, we applied it to a large set of programs, among which are three large, popular and high-performance applications ported to leverage TEE-facilities —*i.e.*, an HTTP telemetry tool that TEE-offloads its sensitive analytics, a packet-filtering tool that TEE-offloads its packet matching facilities, and a VPN-as-a-Service that TEE-offloads its encryption and decryption phases. The effort of porting these applications is lowered significantly by LuaGuardia’s abstractions and services. The overall overhead due to the TEE averages 13– 41% (avg: 18%). Applying several optimizations lowers the overheads to 5–95%, especially for short-lived program fragments. Summarizing, the contributions that LuaGuardia offers are the following:

- We design and implement LuaGuardia, a lightweight code offloading system centered around Intel SGX, namely LuaGuardia, that enables confidential computing using a high-level language, such as Lua, eliminating the need to learn or port code to device-specific TEEs.
- We perform several low-level optimizations as well as implement additional features, such as enclave pre-allocation, capture memory snapshots and system-call batching, to accelerate LuaGuardia’s runtime performance, boosting the overall performance by up to 70%.
- We show how LuaGuardia can be used by a diverse set of applications, from cryptographic algorithms and simple popular benchmarks to real-world networking toolkits. Our evaluation results show that LuaGuardia has an average overhead of 18% for real-world applications, compared to their unprotected counterparts.

1.1 Guarantees

LuaGuardia offers and ensures the following strong guarantees for each new Lua script that is executing within LuaGuardia’s interpreter:

- Operations on the given code, data and modules only happen once. After that point, the execution is self contained inside the enclave with no interaction with the untrusted world (providing that no call outside of the enclave occurs).
- The communication channel between the LuaGuardia Client (Section 4.1.2, 4.1.3) is end-to-end encrypted, thus all the exchange data, code and results are never exposed neither to the network operator nor to a malicious cloud provider. Same logic applies to the handling of I/O system-calls such as `fread/fwrite`. Any potential read, will first perform a round of decryption on the contents of the file, and then process them inside the enclave, whereas a write will encrypt the data before exiting the enclave.
- LuaGuardia ensures that even if the code executing inside the trusted interpreter is hostile to the enclave, the sandboxed environment that is provided will prevent it from interfering with the normal operation of the system.

- By leveraging the attestation primitives that Intel SGX [8] offers, a client may safely gain the trust of LuaGuardia server about the validity, the soundness and the validity of the enclave running her code. Furthermore, at post execution, apart from the results, LuaGuardia server returns an execution log with the hash of the code, data and input that executed as an extra token of reliability.

1.2 Outline

The rest of this dissertation is organized as follows. Chapter 2 presents a brief background on the most used and popular Trusted Execution Environments available in the market. In the same chapter, we also describe in detail why did we select Lua language among others interpreted languages and the benefits it offers to the user. In the next Chapter 3, we present the Threat Model of our system and the assumptions we make for our work. In Chapter 4 we describe in great detail the core components of LuaGuardia regarding the client entity, the server entity and the main Lua virtual machine.

Moving on, on Chapter 5 we describe the whole porting process of the Lua interpreter inside the SGX enclave, the challenges that we had to deal with and the optimizations that are implemented that ameliorate the performance of the enclave (since TEEs introduce overhead to the application). More specifically, we document how the execution process works, how LuaGuardia handles system-call requests, dynamic support for Lua modules, global state preservation and execution modes.

Chapter 6 provides a thorough evaluation and analysis of LuaGuardia framework on several different micro application benchmarks that range from memory bound algorithms, to I/O algorithms and cryptographic functions on different size of input. Moreover, we present the overhead LuaGuardia introduces to larger scale applications as real use cases. On Chapter 7 we present an analysis on the limitations SGX imposes to our implementation, problems that LuaGuardia cannot handle at the moment and potential attacks that render our framework is susceptible to. On Chapter 8 we survey prior existing work regarding SGX and interpreter languages as well as code offloading and finally Chapter 9 summarizes the dissertation and points out future research directions.

Chapter 2

Background

Developing a program to execute on a TEE poses several challenges. To make these challenges concrete, we describe several example applications (Section 2.3), none of which can trivially be implemented on or ported to today's TEE abstractions. These examples illustrate key requirements for the design of a framework for confidential computing, which we overview next. Before describing these examples, however, we describe in great detail the term of TEE and also briefly refresh the most commonly used TEEs.

2.1 Trusted Execution Environments

The term TEE is widely used among chip manufacturers and platform providers but nonetheless no precise explanation on the term has been explicitly coined. To overcome this issue, we cite several definitions of TEEs showing the discrepancy between the different terms:

- Tal et al, 2003 [9] refer to TEE as: Trusted virtual machine monitor (TVMM) that partitions a tamper-resistant hardware platform into multiple, isolated virtual machines (VM), providing the appearance of multiple boxes on a single, general-purpose platform.
- OTMP, 2009 [10] describe TEE as: An Execution Environment with security capabilities that resists against a set of defined threats and satisfies certain properties related to isolation, secure storage, cryptographic operations and anti-code tampering.
- GlobalPlatform, 2011 [11] refers to TEE as: An environment where the following properties are valid:
 1. Any code executing inside the TEE is trusted in authenticity and integrity
 2. The other assets are also protected in confidentiality
 3. The TEE shall resist to all known remote and software attacks and a set of external hardware attacks

4. Both assets and code are protected from unauthorized tracing and control through debug and test features
- Vasudevan et al, 2014 [12] propose the term as: A set of required resources intended to enable trusted code execution: (i) Isolated Execution, (ii) Secure Storage, (iii) Remote Attestation, (iv) Secure Provisioning and (v) Trusted Path.
 - Sabt et al, 2015 [13] analyze in great detail that a TEE is: A tamper resistant processing environment that runs on a separate kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states (e.g. CPU registers, memory and sensitive I/O), and the confidentiality of its code, data and runtime states stored on a persistent memory. In addition, it shall be able to provide remote attestation that proves its trustworthiness for third-parties. The content of TEE is not static; it can be securely updated. The TEE should resist against all software attacks as well as the physical attacks performed on the main memory of the system.
 - Quarkslab, 2018 [14] refer to TEE as: A secure area inside a main processor that runs in parallel with the operating system, in an isolated environment. It guarantees that the code and data loaded in the TEE are protected with respect to confidentiality and integrity.

As we can conclude from the above various definitions of TEE term, the common points on all of them can be summed up to: A system that is secure, offers isolation and a sandboxing environment independent from the main operating system that supports memory encryption, persistent storage capabilities, trusted code execution that has high levels of reliability, a form of unique identification to accommodate attestation and cryptographic services [15], [16], [17], [18], [19], [20].

2.1.1 Intel Software Guard Extensions

Intel Software Guard Extensions (SGX)[21] is a hardware extension available to recent Intel CPUs, firstly available on Intel Skylake family processors, targeted for x86 architecture that has as a goal to provide system integrity, sandboxing, data protection and confidentiality guarantees to security-sensitive code and data computations performed on a system where all the privileged software and hardware (kernel, OS, peripherals) are potentially malicious that try to disrupt normal operations. Intel SGX introduces the term of `enclave` as trusted execution environment provided by the SGX API to the applications.

Internals The data and the code that reside inside the enclave are protected from the untrusted outside world and are stored inside a memory protected region called `Enclave Page Cache (EPC)`. Kernel, other enclaves and non-enclave applications may not access an enclave's code, whereas an enclave code may access untrusted code (only to the application that is bound with it) by using `Outside Calls`. `Outside Calls` (`ocalls`) refer to the allowed function calls an enclave may perform to the untrusted part of the application. Similarly, one or multiple untrusted applications that are bound with the enclave may perform `Enclave Calls` to access the enclave functions. `Enclave`

Calls (ecalls) are strictly defined functions that reside inside the enclave and act as normal functions. The functions that are allowed for ecalls/ocalls are defined at `Enclave Definition Language (EDL)` which is a script file that defines the functions as well as the contains information regarding the flow of the data and various checks that should be applied on them before any function call connected with the enclave. One of the primitives of Intel SGX is to keep the Trusted Computing Base (TCB) as small as possible in order to minimize the potential attack surface. Currently, the max available memory in EPC ranges from 64 - 128MB [22]. Consequently, only a limited number of active enclaves may be live on the same time on the system. On Windows OS, 128MB is the max available memory that an enclave may support. On contrast, on Linux OS the developer may explicitly request additional protected memory during compilation time up to several GB through paging. When memory pages of the enclave need to be swapped out and moved to the untrusted DRAM, a supplied chip mechanism available called `Memory Management Engine (MME)` is responsible for encrypting the requested pages, and then storing the to the DRAM. Similarly, when swapped pages must be read by the enclave, MME unit fetches the encrypted pages and performs on-the-fly decryption to read the actual content and data.

Application and Tools An SGX application can be viewed as two separate and distinct entities that communicate through a trusted bridge. The untrusted part of the application that is located and is being executed in the non secure system memory and has full access to all the resources available to the platform whereas the trusted code is executed inside the enclave and the EPC that has access to the features provided by the SGX API. SGX applications execute strictly in Ring 3 (user-space) and are supplied with a modified and stripped version of `libc` in the extend that they do not have access to system-calls, I/O or privileged instructions. To implement such calls and requests, the developer has to define them in the EDL file before compilation to create these edge routines. `Edger8r` is the tool that is shipped with the SGX SDK and is responsible for creating such bridges between the untrusted part of the application and the enclave. After the edges have been created and the compilation is finished, the `Enclave Signing Tool` is invoked that produces a signature that contains enclave properties. Any post compilation changes to the code, the data or the enclave signature can be detected and the execution will be aborted.

Life of an Enclave In the SGX API, the instantiation and initialization of an enclave is being carried out by the non secure entity of the SGX application. Specifically, by invoking the `ECREATE` instruction, a free EPC page is converted in SGX Enclave Control Structure (SECS) containing information related to `BASEADDR` and `SIZE`. SECS are the metadata per enclave that are stored by SGX and are tied to each enclave. An enclave's identity is identical to its specific SECS which is the first step to the enclave creation, whereas the destruction of an enclave would result in the deallocation and eviction of the reserved EPC pages. `ECREATE` validates the content of SECS and results in failure in case of invalid values. Additionally, it is responsible to initialize the `INIT` attribute of enclave with the false value, prohibiting the trusted code execution until its value is switched to true as well as setting SECS as uninitialized. In the current state, `EADD` instruction is issued by the system to load additional code and data in the enclave but also to create new EPC pages. `EADD` reads parses a `Page Information Structure (PAGEINFO)`

which acts as an intermediate to communicate with the current SGX implementation. The information contained in PAGEINFO are the Virtual Address (VA) of the EPC page that will be created, the VA of the non secure pages that will have their data copied into the EPC page as well as the permission attributes of the page. In the current version, changing the permissions of EPC pages is not supported by the API. After the data and code has been loaded to the enclave, the system makes use of a Launch Enclave (LE) to acquire a unique EINIT Token Structure bound to enclave that marks the SECS state as initialized. LE is a privileged enclave provided by Intel and is a requirements to instantiate and launch new enclaves. After LE is launched, enclave's INIT field is set to true, meaning that execution may now start on Ring 3. After this point EADD may no longer be issued on the same enclave. The untrusted program may now issue EENTER instruction to execute enclave code. When the execution of a trusted function completes, EEXIT is issued to context switch from the trusted to the non secure part and the results are returned to the caller. An enclave is destroyed only when all of its' EPC pages are deallocated and the SECS page is freed. This is done by the EREMOVE instruction. A normal trusted function execution is shown at Figure 2.1.

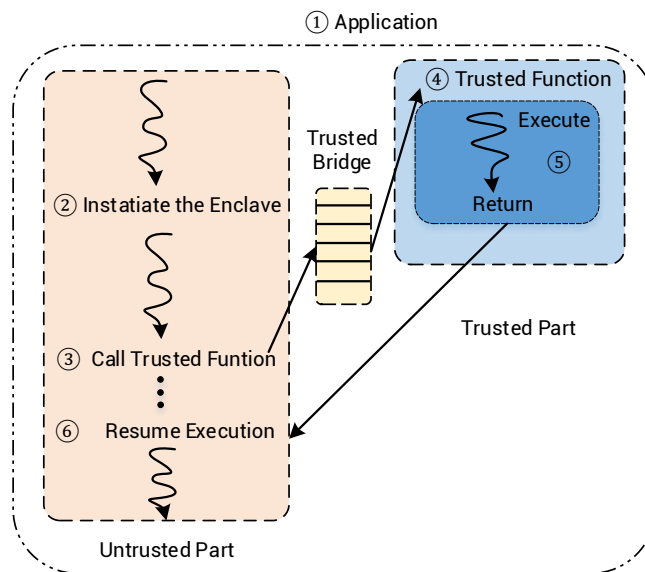


Figure 2.1: A normal trusted function execution. The untrusted requests the creation of the enclave and then invokes the requested function. This request is forwarded to the Trusted bridge that performs certain checks and validations, and then passes the request with its arguments to the respective function inside the enclave. The code inside the trusted part is executed inside the EPC cache and the results are fetched back to the call site. The code then resumes its normal execution.

2.1.2 ARM TrustZone

Similar to Intel SGX, ARM TrustZone [23] is a set of hardware extensions available in almost all ARM CPUs. The benefit on TrustZone is that is available in the majority of mobile phones, smart devices and IoT boards making the most widely available TEE in the market [24]. TrustZone denotes two separate worlds: the Rich Execution Environment (REE) which is called as normal world and the Trusted Execution Environment (TEE) called trusted world. To ensure complete world isolation, TrustZone offers hardware security extensions components such as CPU, memory controllers and peripherals. In contrast to the term enclave, ARM TZ introduces the term Trusted Application (TA) and refers to applications running in the secure world. At any given point during system execution, the world the processor currently executes is determined by the value of a processor bit, known as Non-Secure(NS) bit. The value of this bit may be read by the Secure Configuration Register(SCR) and is transmitted to the whole system. TrustZone introduces a new processor mode that intends to preserve the processor state across world switching and transitions. This processor mode is called `Monitor Mode`. When a secure monitor call(`smc`) instruction is issued, the processor can context switch between the secure and the normal world and freeze the execution in normal world application [25]. In contrast with x86 Ring execution levels, ARM architecture has Processor Modes called Exception Levels(EL). Armv8-a/AARCH64 offer four exception levels and two security states (secure or non-secure bit). EL0 is the lowest privileged execution level where normal applications run in user-space. EL1 is the kernel space, normal kernel if the code executing is running on non secure world, or trusted kernel, which is a standalone kernel that executes in secure mode that handles requests for the TAs. EL2 is reserved for the hypervisor(Secure Partition Manager for the secure part that configures security attributes for the peripheral devices [26]) and EL3 is used for Secure Monitor [27]. A clear overview of the EL states is shown at Figure 2.2.

Despite TrustZone's wide availability on IoT devices and phones, several problem it faces do not encourage further research and development on this field. First of all, due to TrustZone nature, the existence of two separate worlds poses a huge threat to the integrity of the system. In case one of the TA's running in the secure world becomes compromised, all the other TA's are susceptible to be attacked as well since they share the same memory and address space. Furthermore, even though the underlying hardware the TrustZone extensions exists in almost every ARM processor in the market, there is no universal SDK/API by ARM. As a result, there are several independent and distinct SDKs([28], [29], [30], [31], [32]) targeted for different devices and boards that in most cases are closed source and not a available to the end user. Due to high costs of the secure memory in TrustZone, the available live memory to the end user ranges from 3-5MB [33], which is expected to have small memory footprint in the trusted environment, reducing the attack code surface, but also limits the available resources to the developer.

Some of the security features that Arm TrustZone provides are: (1) Security States (Secure thread, Trusted Code Handler, Library Managers, RTOS), (2) Secure interrupts, (3) State transitioning (boundary crossings), (4) Memory management, (5) Trusted boot and (6) Secure code and data memory [34]. In contrast with Intel SGX, TrustZone lacks

the support for providing a secure root-of-trust and attestation services, thus rendering the system unable to attest to the user or an external verifier that the code running in the isolated processor is trustworthy [35]. So it is up to the chip vendor to provide additional hardware to amend for the lack of attestation features [35], [36].

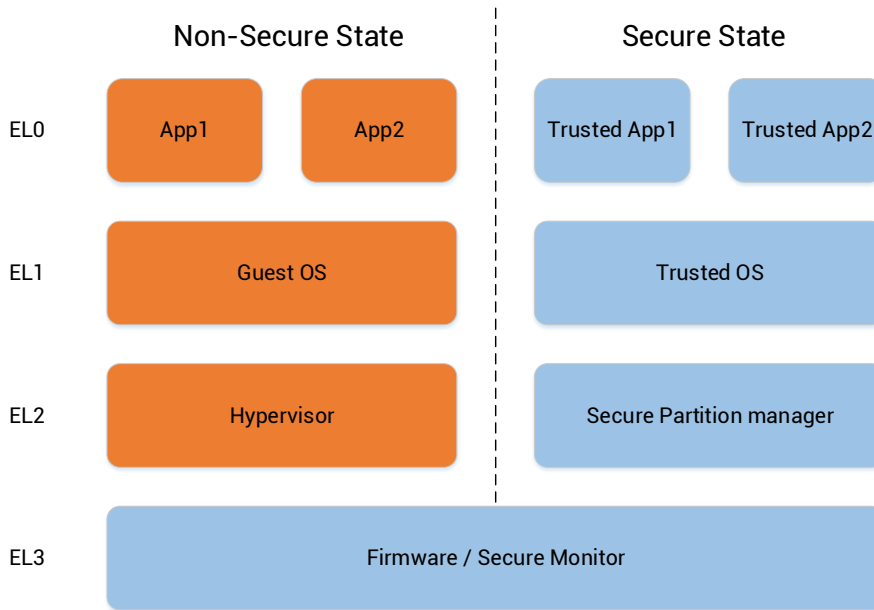


Figure 2.2: Exception Levels and Security states on Armv8-a architecture.

2.2 Just-In-Time Languages

2.2.1 Why Lua

Lua [37] is a powerful high level, lightweight, portable and strongly typed scripting language. It offers several state-of-art programming features such as procedural programming, sandboxing, object-oriented programming, functional programming that is targeted for embedded uses in applications. Lua has been utilized in many industrial and commercial programs and applications such as Adobe Lightroom [38], Wireshark [39], Snort [40], World of Warcraft[41]. Lua is portable and cross-platform since it's interpreter is developed in ANSI C that may be built out-of-the box using a compliant compiler. Lua's small memory footprint(40K) as well as small code size (150-200K) [42] makes it an ideal participate for Intel SGX limited resources.

SGX SDK official API is documented on native C/C++ making Lua a perfect candidate since it is written in plain C. At the time of development of LuaGuardia, Lua5.3 version [43] was the latest revision of Lua. Its size is about 19K LoC and it consists of

20 source code files, which is a pretty small cost for the feature it provides [44]. Moreover, Lua provides a fast and efficient built-in Garbage Collector(gc). The collection process initiates automatically and cleans all the objects that are considered dead, unreachable and are no longer needed by the execution state. Lua implements an incremental mark-and-sweep collector and is based on two fields that control its collection cycles: (i) `garbage-collector pause` which controls the waiting time of the gc before initiating a new cleaning process and (ii) `garbage-collector step multiplier` that controls the speed of the gc in relation with the memory allocation. Since both of them are easily configurable, the developer of an enclave is facilitated since the memory de-allocation happens automatically and all the valuable resources are freed, without the live EPC pages, resulting in swapping them to the non-secure DRAM and degrading overall enclave performance.

Dynamic module loading and support is another great benefit for Lua. Specifically, it is quite easy to interface Lua with the C API as well as write packages in C language targeted for Lua programs. A developer may leverage and use the C API provided by Lua source, and program dynamic libraries that may be loaded during execution time of a Lua program but also access Lua internal structures and entities prior unavailable to the traditional Lua execution environment. Apart from dynamic libraries, one could provide Lua programs with library modules[45]. Modules are a core part of Lua language since they extend the functionalities of a program. When a module is executed, all its contents (functions and statements) are executed and then are imported to the global environment (all functions, tables and global variables are stored to `_ENV` or `_G` variable, which is a global-scoped table storing the environment of the Lua execution state). In LuaGuardia, Lua module support is fully supported 5.3 whereas C module loading is unsupported due to SGX limitations 7.2.

2.3 Motivating Examples

Consider a few confidential computing scenarios where part of an application needs to run on untrusted, TEE-enabled devices. One such example is an HTTP telemetry tool that can be deployed in SDN/NFV or 5G environments to gather periodic performance statistics into a TEE where it runs analytics—*e.g.*, to extract average latency or detailed latency percentiles. Another example is a packet-filtering tool that offloads sensitive packet filtering tools and associated matching to the TEE. A final example is the ad-hoc establishment of secure connections between private networks and public cloud providers (*e.g.*, VPN-as-a-Service) that uses the TEE to protect the key establishment with the remote peers and the corresponding cipher operations.

In all these examples, a TEE offers critical security benefits to the part of the application that operates on sensitive data in untrusted environments. Unfortunately, reaping these benefits requires the manual development or porting of TEE-executing fragments using low-level interfaces provided by the TEE manufacturer—which causes several practical challenges.

Manual Effort Developing in a low-level language requires significantly more effort and

care than developing in high-level languages—an effort that is even more pronounced for short program fragments for analytics or pattern matching like the ones described above. Even if the program is already implemented in a low-level language such as C/C++, which would simplify the use of TEE API, it still requires manual partitioning, re-compilation, library porting, and linking of the entire application, including the components running outside the TEE.

Extensibility In many domains such as analytics, learning, and telemetry, there is a need for extending or updating a program during its execution. This causes an impedance mismatch with the static nature of TEE-executing code where adding new functionality dynamically or executing code is impossible. In these cases, adding even a single-function module requires compilation, linking, and re-deployment of the entire TEE-executing component—including starting the TEE afresh. This entire process is on the order of tens of seconds, rather than tens of millisecond required for simply shipping a function to the TEE.

Safety Finally, by rewriting the analytics component in a low-level programming language such a C/C++ makes the analytics program susceptible to traditional memory corruption attacks—after all, memory and type safety and their affects to security are a key reason why developers use a high-level programming language.

2.3.1 LuaGuardia Overview

To address these challenges, LuaGuardia embeds a Lua runtime inside a TEE to address the challenges of manual effort, dynamic extensibility, and runtime safety—by piggy-backing on the characteristics of a high-level programming language. Lua is a general-purpose embeddable programming language, offering memory and type safety, data-description facilities, and runtime meta-programming—including ones that allow a program to manipulate its own environment [46].

Manual Effort LuaGuardia offers significant developer economy compared to low-level abstractions, because of the productivity benefits stemming from a high-level, dynamic programming language. Our example applications can leverage the Lua library ecosystem, without having to develop them from scratch—such as analytics, learning, and inference. We note that LuaGuardia does not require the entire application to be written in Lua: as Lua is embeddable in C/C++ programs, LuaGuardia can be used to create a high-level TEE interface even for programs developed in low-level programming languages— and we show this in our evaluation section.

Extensibility Based on a dynamic language, LuaGuardia is naturally extensible via runtime code evaluation of Lua code. Using LuaGuardia’s interface at runtime, the engineer in our example would be able to extend and configure the available runtime functionality by sending and evaluating additional functions during the execution of the platform.

Safety Finally, as Lua is a memory- and type-safe programming language, the engineer does not need to worry about low-level attacks in their code. LuaGuardia provides additional wrappers to limit the access of TEE-executing code to key interfaces, offering a lightweight sandbox that provides additional protections beyond the base type- and

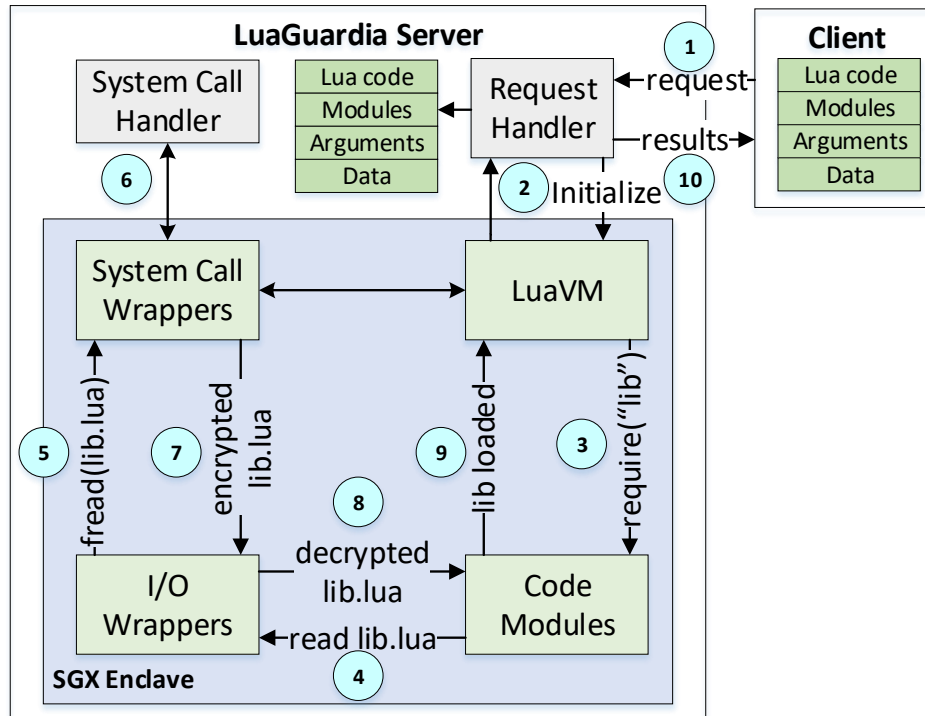


Figure 2.3: LuaGuardia overview

memory-safety guarantees provided by the Lua environment.

While the current enclave programming models provide sufficient code security and data confidentiality, often times they do not appeal to developers creating modern and modular applications for several reasons. For example, applications that wish to provide a main functionality that can later be extended with modules and plugins cannot utilize secure enclaves at their current state. The first limitation is that enclave-protected code has to be statically compiled and signed during development and any subsequent modifications to their code base requires the re-compilation of the entire source tree. Moreover, the available abstractions offered by the enclave infrastructure are low level and require experience and proper understanding of their semantics in order to provide sufficient security. However, even when properly utilized, enclave protected code has to be developed either in C or C++ which do not offer type and memory safety, thus, implementation bugs can still compromise the application's security. One could argue that a system that provides a certain degree of extensibility without the need for complete source tree re-compilation could be developed from scratch. However, such systems usually result to big code bases that break the as small as possible TCB design principle encouraged in TEE development.

Chapter 3

Threat Model and Assumptions

LuaGuardia assumes a powerful and malicious entity, possessing super user privileges, has full physical access to the underlying hardware and peripherals, the whole software stack, the Operating System as well as control the kernel but cannot interfere or tamper with the CPU. Moreover, we consider denial-of-service attacks (DoS) on SGX software, libraries and hardware, access to network and file-system out of scope for our work. Additionally, an attacker could potentially halt or interrupt the execution of LuaGuardia, however that should not enable her to extract or guess any sensitive enclave secrets, encryption keys or data. Furthermore, attacks targeting the Intel SGX hardware such as side-channel, timing attacks or cache attacks ([47], [48], [49]) are orthogonal to the implementation of LuaGuardia and any potential research work that manages to counter and mitigate these kind of attacks may have direct impact and benefit on our system. LuaGuardia assumes that the design, implementation and API of Intel SGX SDK is free of memory bugs, leaks and vulnerabilities. Finally, we assume that all the components and entities (Section 4.1) of LuaGuardia are free of software vulnerabilities that could compromise the system's integrity and security.

Chapter 4

System Architecture

4.1 LVMAT Components

LuaGuardia is an ecosystem of several components (trusted and not) that co-exist and their final goal is to achieve trusted code execution. The architecture of LuaGuardia is shown in Figure 2.3. Typically, an application that runs in the untrusted domain aims to offload security-sensitive computations to the trusted component, whereas the latter is responsible for accepting and handling secure code execution requests encapsulated within SGX enclaves.

4.1.1 LVMAT Interpreter

As the name suggests, the whole interpreter of Lua language had to be ported inside the SGX in the enclave as the only trusted and secure part of the whole framework. By doing so, we ensure that our interpreter is trusted at pre-compilation phase, signed by Intel during the compilation, thus rendering any potential code injection or data or code manipulation attacks or actions impossible. Due to the limitations of the SGX API and the limited resources it provides (since it acts as a reverse sandbox), several changes had to be done on the vanilla Lua implementation. First of all, all the signal dependent code has been removed since the required headers are not available nor may be provided in enclave libraries. Moreover, references to `dlopen` family of functions are not supported by Intel SGX since they can impose a great risk to enclave integrity. This occurs due to `dlopen` dynamically loading and mapping libraries on the memory, and since `dlopen` is a system-call, it has to be proxied to the untrusted part of the application to be served. As a result, the mapped memory will be allocated in the untrusted RAM, thus none of the potential native Lua modules can be trusted, so such functionality is opted out. To be honest, enclave does not offer any I/O functionality neither any system-call handling. In a real world scenario where a malicious entity has full control to our computer, we cannot trust any component rather than the underlying hardware provided by SGX, so system-calls are disabled. To overcome this issue, there are two available alternatives. The first is to provide proxy functions and requests to the untrusted environment. This is done by crafting and declaring the requested helper functions in the EDL file (pre-compilation

phase) and implementing them in the trusted part of the application. Additionally, these proxies have to be implemented in the untrusted part in order to perform the requested untrusted calls such as system-calls, I/O, network operations. The most crucial part of the interpreter is the `FILE` structure and the functions such as `fopen`, `fread`, `fclose`, `fgets`. These functions are important for fetching and reading the code to be executed inside the enclave, writing the program execution results to files, returning them back to the client and providing basic file functionality, prior unavailable to the enclave. As mentioned before, neither these function calls are trusted and their results may be altered. However, on Section 5.2 we describe in great detail on how we handle those untrusted calls and validate the integrity. The second alternative to handling system-calls is to emulate them by offering a self-contained software implementation of the system-call based on the existing API provided by Intel SGX SDK in combination with proxied function calls. Let's assume that `read` system-call is exposed to our application through EDL interface, thus any function call that handles file reads(`fgets`, `fread`, `fgetc`, etc) may be built upon `read` system call without increasing the TCB of our application and minimizing the number of OCALLs to the untrusted world.

4.1.2 LVMAT Server

The component with the most significant value and importance in the whole ecosystem is the LuaGuardia server. In the trusted environment, LuaGuardia provides a language virtual machine, based in Lua. It primarily consists of an SGX-enabled Lua interpreter which is responsible for performing computations and has no access to system-calls or other available peripherals. Such additional functionality that requires to access the untrusted environment as well as components of the Operating System, but also handling of incoming client connections are offered by the driver code of the enclave. In this way, we manage to limit the TCB of our system, while being able to offer on par functionality and features with the original Lua implementation. On Chapter 6 we evaluate our base implementation with the vanilla Lua interpreter. The server application is responsible for several tasks that are required in order to ensure and instantiate trusted end-to-end communication between the two revolving parties, the client that requests to execute code computations in a trusted environment as well as the LuaGuardia server that handles the code execution requests and serves the user. First of all, the server awaits for incoming code execution requests. As soon as a client connects to the server, both parties must perform a Diffie-Hellman [50] in order to generate the shared key for the encrypted communication. The steps required for this operation to complete are the following: (1) Generate key-pair of public and private keys inside the enclave and forward only the public key to the untrusted environment, (2) send the public key to the client, (3) receive the public key of the client in the untrusted app and store it inside the enclave, (4) generate the shared secret that resides inside the enclave using the private key of the enclave and the public key of the client, (5) send the encrypted shared AES key to client. Similar steps are also done on the part of the client. As soon as the previous steps are completed, all the traffic between the two entities will be encrypted using the shared AES key. After this point, the server receives the requested code, modules and data input all in encrypted format. These

data are passed through to the enclave, where the decryption phase takes place, gets the plain data and start executing the requested code inside the enclave interpreter. After the execution is done, all the data results are encrypted, proxied to the server socket which is responsible for transmitting the data back to the client.

4.1.3 LVMAT Client Stub

The client component of LuaGuardia acts as a standalone thin layer library. Firstly, it enables remote users to securely connect with LuaGuardia server, over an end-to-end client-server-enclave encrypted communication channel and request the execution of Lua scripts in a protected memory space. The simple thin layered design of the LuaGuardia client, enables clients to offload new as well legacy Lua applications to the enclave interpreter with minor changes to the existing Lua applications. Furthermore, LuaGuardia client does not require SGX capabilities and its most critical task is has to deal with is handling and establishing the encrypted communication channel with the server's commodity cryptographic operations that may either be hardware capabilities apart from SGX, or software implementations. As a result, this enabled the LuaGuardia client to operate on low power or embedded devices.

Similar to the LuaGuardia server, the client has also to perform a certain amount of steps before performing a secure Lua code execution. (1) Generate a new key-pair and transmit the client public key to the server, where it requests dynamically a new code execution environment. (2) The server is informed about the incoming request and a Diffie-Hellman key exchanged is performed between the two entities. (3) The client, the key establishment is performed entirely in the untrusted memory since we assume that the client does not have TEE capabilities, although a Trusted Platform Module (TPM) may be used for this step, whereas on the server side, the key generation is entirely generated inside the enclave. (4) The newly-generated keys are stored inside the enclave and are never exposed to the untrusted DRAM nor the file-system. This process generates a client-server-enclave communication channel that prevents malicious users from eavesdropping and monitoring the network, server's memory or the file-system to obtain the exchanged data. (5) Once the communication phase is done, and the unique AES key is shared between the two parties, the client may start offloading the Lua code with all its dependencies (modules, input data) and encrypt them using the shared AES key. (6) Finally it awaits for the server to send back the results.

4.2 LuaGuardia Local Execution Mode

Another mode that is supported on LuaGuardia server offers local code execution where the client and the server are co-located on the same physical machine. This option enables newly developed or legacy Lua applications to be deployed and make use of SGX capabilities without the need of transmitting code over the network. However, we must note that this mode is opted to use under several conditions. First of all, this mode may only be considered secure in case where a malicious attacker has absolutely no control over the

system or the file-system, but her influence restricts to compromising user space applications. The main problem for this relies on that LuaGuardia's virtual machine has to fetch the executable code, loadable modules and input data from the untrusted file-system or the DRAM. Since no prior encryption has been performed, the files are stored in plain-text format in the untrusted memory. Adding an extra layer of encryption to the files is futile because the operation will take place in the untrusted component of the application. So, by controlling the file-system, a malicious party may compromise and leak the Lua code, data and input while they are being transferred into the SGX enclave.

Chapter 5

System Implementation

In this section we are going to present and demonstrate the full implementation details of LuaGuardia as well as the challenges we encountered and optimizations we implemented while porting the Lua interpreter inside the SGX enclave.

5.1 Porting the Lua VM

The interpreter we based our implementation on is Lua 5.3.5 [37]. Compiling the vanilla Lua source code generates two distinct binaries. The first one is named `lua` that is capable of loading and executing plain Lua scripts or pre-compiled Lua binary chunks. The latter output is named `luac` and it is the Lua compiler, responsible for translating Lua source code into binary files that contain pre-compiled chunks that can later be loaded and executed by the interpreter at a different point in time. We chose to utilize and port the official Lua virtual machine instead of the third-party developed LuaJIT [51], despite the latter offering considerable performance benefits and increase, for several reasons. First of all, using the official Lua interpreter guarantees that all the features provided by the language will always be up to date, according to its standards and ISA, thus offering almost full compatibility with vanilla Lua virtual machine. Secondly, LuaJIT offers a bigger TCB, a property that we strive to keep as small as possible for memory efficiency and auditing reasons. Moreover, offers Lua script compatibility up to version 5.1, which is the latest available version of LuaJIT, so any newer script of Lua will not be compliant with its environment. Finally, SGX enclaves do not support the creation of additional executable memory pages during execution, a feature, that will be available on version 2 SGX hardware revision [52]. Our LuaGuardia implementation utilizes the original Lua implementation instead of the Lua compiler as our main interpretation engine that will be running inside the execution environment. The original code base of Lua consists of 25 headers files, containing 4269 LoC, and 35 source code files containing 19482 LoC of native C code. The total code base spans 23751 LoC, rendering the Lua language a perfect candidate for a protected dynamic language environment as it keeps a minimal TCB that can easily audited. We port the vanilla Lua virtual machine into the SGX enclave on an Arch Linux based host running kernel version 5.7.6 using the official Intel

SGX SDK version 2.8 for Linux and the used for compiler GCC 9.2.1. The whole LuaGuardia implementation consists of 26132 LoC, where 1377 LoC comprise the untrusted part of LuaGuardia while the rest 24755 LoC comprise the trusted part of the application, residing in the trusted environment.

While our port tries to keep a quite minimal code base, the implementation of the interpreter is not a straightforward task. First of all, the original code had references to `signals` and `dlopen` family of functions that are currently not supported nor provided by the SGX SDK API. For these reasons, we perform several changes and modifications to the original code to bypass these requirements without having a negative impact nor affecting the expected functionality of the virtual machine. Moreover, the Lua source code contains and requires several function calls in order to operate normally. Some examples of them are `fopen`, `fwrite`, `clock`. Of course, none of these calls are neither provided or exposed by the SDK API since all of them result into system-calls, which are excluded from the SGX environment. To solve this problem, we develop and provide wrapper functions in order to proxy such unsupported function calls to the untrusted part of LuaGuardia's component, where they are served.

Except from the interpreter port inside the enclave, we also have to provide the required functionality in the enclave in order to enable LuaGuardia to act as a remote server and service, to establish a secure communication channel with its candidate clients. Moreover, the server has to be able to handle and parse encrypted blobs of client requests containing Lua code, Lua modules and input data that all of them are going to be forwarded inside the enclave, decrypted and finally executed within the interpreter. Finally, we develop a protected instance manager, that is responsible for handling the various clients and providing each one with a separate instance for isolation purposes.

5.2 system-call Handling

The most common issues with many TEEs, including Intel SGX, is the lack of system call support. This is normal and expected, since in any real-life SGX scenario, the only piece of software and hardware that are considered trusted is the same the enclave and the underlying SGX hardware, thus system-calls, that are not considered components of the trusted environment, are excluded and not supported. Assuming that the enclave has access to system-calls and that the kernel is malicious, such direct call from the enclave to the kernel may compromise and expose enclave secrets as well leak protected data, thus rendering the whole security scheme provided by SGX hardware useless. On the other hand, almost all applications with SGX capabilities need invoke system-calls to perform basic functionalities, such as access to peripherals, access to the network infrastructure or the file-system. When utilizing Intel SGX, most of the times these calls are requested from within the enclave code and are forwarded through proxy functions (provided in the EDL file) to the untrusted part of the application, where they are served and handled by the system. In particular, it is up to the developer of the enclave to design and implement that intermediate layer that will handle and bridge the enclave code with the untrusted application that will handle the requests and fetch back the results from the untrusted part

of the application.

Normally, custom built SGX-enabled applications might require only a few system-calls that can be easily wrapped by the developers, however this same scheme does not apply to LuaGuardia. Since our implementation leverages and utilizes a full-fledged, drop-in replacement of Lua virtual machine, enclosed in SGX enclaves, several challenges have to be addressed. First, we need to accommodate the system-calls that are required by the Lua virtual machine itself in order to function properly. Second, new system-calls may be issued during the execution of different Lua scripts. The former case may be considered a more straightforward approach, as the required system-calls can easily be accounted for, which unfortunately does not apply in the latter case where all the system-calls requirements may be known a-priori but will be resolved at execution time.

The most common approach to handle such an issue, is to implement custom system-call wrapper for each and every one of them. However, modern operating systems provide several hundreds of system-calls (about 400 on Linux), without the majority of them being necessary required or called by the Lua interpreter or an executing Lua script. Furthermore, many of these system-calls, could be triggered by an offloaded script and abused to perform malicious activities on the remote host. For this exact reason, we decide and provide interfaces for only the bare minimum of the system-calls that are required in order to achieve normal functionality of the Lua interpreter inside the enclave but also to minimize the potential attack surface without having to keep proxying functions to the untrusted application. By observing the original source code, we realized that the most commonly used functions, resulting to system-calls, are those that are associated with data I/O such as `fopen`, `fwrite`, `fread`, etc. In order to provide system-call support to the Lua interpreter residing within the SGX enclaves, we have to develop custom functions that will be defined in the EDL file and implemented in the non-enclave part of the application, performing an OCALL to it for each pending request. Once such a request is handled to the OCALL function, since the untrusted part of the application has full access to the whole system, the handle will be handled, get the results, and fetch them back to enclave after performing the typical SGX checks to the data. However, the validity of the data transferred from the outside of the enclave within the enclave have to be explicitly checked for their integrity the mechanisms that have to be implemented by the application developer. In the SGX model, the only entity we assume trusted is the enclave, whereas the Lua scripts, Lua modules and input data that are required for the execution, must be provided with the each client request in encrypted format, thus we prohibit read and write operations to arbitrary file-system locations at the server side. Optional file offloading and handling may be performed by the client in predefined server file-system locations in an encrypted manner, using SGX's sealing and unsealing functionality offered by Intel's API, enabling secure and persistent storage in the untrusted file-system. A list of the supported functions resulting in system calls available to the protected Lua interpreter is presented in Table 5.1.

Table 5.1: Most used functions supported by LuaGuardia

Ported Functions				
fopen	freopen	fclose	ftell	fseek
fread	fwrite	fgets	fputs	feof
ferror	exit	clock	getenv	time

5.3 External Modules

By design, Lua is an extensive programming language that offers dynamic module capabilities in two ways. It has built-in support for two types of libraries: (i) libraries that are implemented in entirely pure Lua code, and (ii) libraries that are native modules built in C/C++ that may be registered using the native Lua C API. In order to support the former module loading, there are two main properties that are required to be supported inside the enclave. The first one is the `FILE` structure in order to handle files and I/O. Secondly, all the functions that are invoked by the Lua virtual machine such as `fopen`, `fread`, `fwrite`, etc are the bare minimum required in order to fetch and load a Lua encrypted module from the untrusted file-system, decrypt it in the trusted part, verify the content of the module against a known checksum for integrity purposes, finally, the interpreter will load the module and start executing it. This specific design prevents attacks from modifying or replacing modules found in the host system's memory/file-system or modules received over the network. The checksums required for module validation may be either pre-stored in LuaGuardia's enclave, such as checksums for standard libraries, or received in encrypted format during the data transferring phase. With this mechanism in place, any `require (<module_name>)` snippets found in a Lua script, import the declared library assuming that it exists in the untrusted file-system (encrypted), and then the module's API is exposed to the developer. This intermediate layer is transparent and offers the same functionality as the vanilla Lua module loading.

On the other hand, native C/C++ libraries, are compiled as shared objectives using Lua's C and may only be loaded by LuaGuardia's untrusted part, using the `dlopen` family of functions. This functionality prevents us from providing dynamic C/C++ library support for the following reasons. First of all, since shared objects cannot be loaded in LuaGuardia's enclave, their integrity may not be verified by the system. Secondly, the functions loaded in LuaGuardia's untrusted part may only interface with the enclave code via a predefined -at compile phase- proxy layer(EDL). Thus, each function's prototype is not known during the execution of Lua code, something contradicting with the design of Intel SGX. Additionally, since the shared objects are stored in the file-system, and `dlopen` family of functions are not available in pure enclave code, a malicious entity may tamper and intercept with their code.

5.4 Maintaining Global State

Typically, the lifespan of Lua script that is executing inside LuaGuardia's trusted environment can be divided in the following phases: (i) setup the end-to-end client-server-enclave encrypted communication, (ii) encrypt and transfer the Lua code, Lua modules and additional data, (iii) the subsequent code execution within the enclave, (iv) the gathering of the results that are sent back to the client in encrypted format, (v) termination of the execution session, cleanup of the resources and reset LuaGuardia to a clean state ready to accept new requests.

The previous phases describe a basic execution example, that does not require to keep any state across program executions. By the end of each LuaGuardia's session, the current state is set to clean, and is ready to handle any new incoming code execution requests. However, it could be quite beneficial for the users to transfer the execution state of a Lua program either from client-to-enclave or vice versa. In greater detail, the developer may explicitly partition the code she would like to execute, declare with tags the critical functions, snapshot the memory state up to the critical functions, encrypt them and wrap them in a blob, and then transfer the execution to the enclave. Switching to the enclave, the snapshotted state of the client's memory is loaded into the newly created environment which is stored in a table called `_G` that is assigned during a creation of a new `Lua_State` during each new Lua interpreter startup. At this point, after the memory is loaded into the table, all the clients values are loaded, and the execution may resume to the critical only functions, with all the required variables and dependencies already loaded in the memory. All this functionality is provided by LuaGuardia's API: (i) the `LuaGuardia _dump` which traverses the environment table and extract all the loaded variables and functions, (ii) the `LuaGuardia _parse` which takes as input a file in JSON format, created by `LuaGuardia _dump` function, and injects the found entries in `_G`. Moreover, `LuaGuardia _dump` is responsible for the encryption of the exported data as well as the creation of the required metadata for their validation. Similarly, `LuaGuardia _parse` decrypts the given file and performs integrity checking on the exported data prior to restoring them to `_G`.

5.5 Code and Client Isolation

By leveraging the hardware and security properties offered by Intel SGX enclaves, LuaGuardia guaranties that a protected Lua interpreter that resides within the enclave stays and remains isolated from the rest of the system, including other untrusted applications, other enclaves, hardware peripherals as well as the operating system kernel. In order to assure and enforce client isolation, LuaGuardia registers a new SGX protected Lua interpreter, residing in a separate enclave, for each different client. As mentioned, before, at each new client session, a new end-to-end channel is established, with a different and unique AES key used for the encryption, that can be optionally preserved across different sessions with the same client. As a result, attackers cannot gain access to the protected resources that reside inside the SGX enclave nor on the LuaGuardia instances utilized by

other clients, aiming to tamper the executed scripts or monitor the execution results.

5.6 Optimizations

One of the main challenges with the implementation of LuaGuardia is providing support for protected Lua script execution within reasonable performance overheads. While the steps described in the previous sections enable the execution of the protected Lua virtual machine inside the SGX enclaves, they are not optimal in terms of performance.

Initialization As it is already described in the previous sections, each new client request for trusted and dynamic code execution establishes a new client connection and triggers the instantiation of a new isolated and protected Lua interpreter instance at the LuaGuardia server. The process of the initialization can be previewed at two steps: (i) issue the creation a new SGX enclave in the untrusted part of the application, (ii) reset the encapsulated enclave to a clean state, ready to handle a new client connection but also free and cleanup the resources and destroy the enclave on the untrusted part of the application. The main problem to this phase is that the creation and the destruction of the enclave introduce a great overhead to the script execution. On Figure 6.7, we can observe that this design may significantly increase the overall performance as the main initialization overhead it derives from the creation and destruction of SGX enclave. Moreover, this overhead is increased as the chosen maximum enclave memory is expanded via LuaGuardia's configuration during the compilation phase.

System-call Batching Since optimizations in the network stack, such as TCP configuration or data compression, are not universally applicable as they benefit a portion of applications, whereas might decrease the performance of others. Another we are looking into to improve the performance and decrease the slowdowns that are occurred by Intel SGX due to the enclave crossings, data validations and pointer checking is to batch system-calls. The motivation behind this optimization is due to the large slowdowns introduced to the handling of consuming I/O operations, due to the fact that SGX enclaves do not have direct access to system-calls. Consequently, every time trusted code tries to invoke a system-call, this invocation is proxied to the untrusted environment based on the EDL file, and is served by the respective function there and the results are returned back to the enclave. After extensive research and evaluation, we found out that the continuous context-switching between trusted enclave code and the untrusted environments performed several validation, integrity and size checks to the copied buffers that decayed the overall performance of the system. In many cases, these issues may be ameliorated by batching a number of the same system-calls. By doing so, system-calls that are not dependent to one another, may be buffered and served with a single, or in worse case, at least much fewer, `OCALLS` in certain parts of the Lua script's execution.

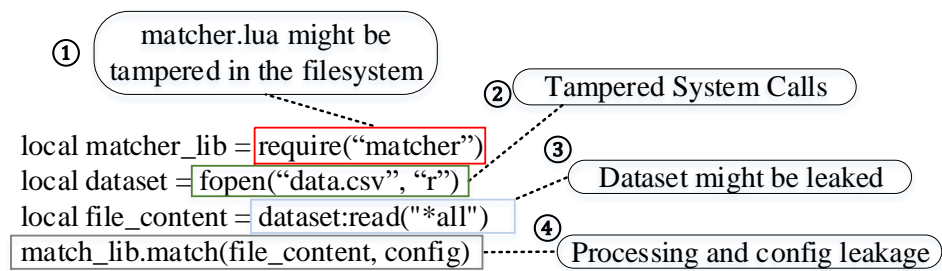


Figure 5.1: Potential risks on running a simple Lua script on the untrusted environment. In LuaGuardia, all the untrusted I/O is handled via encryption, libraries and data are stored encrypted in the untrusted file-system and the attacker may not extract any useful information by observing the enclave code.

Chapter 6

System Evaluation

In this section we present the performance and security of evaluation of LuaGuardia. On the first section, we describe possible attacks against that target LuaGuardia and discuss in great detail on how our system and architecture is able to defend itself against them. Moving on, we provide a thorough performance evaluation and comparison using a series of different and diversified micro-benchmarks, as well as three popular Lua-based real-world applications, namely `wkr2` [53], `pflua` [54] and a custom lite IPsec on top of the `snaab` [55] framework.

6.1 Experimental Testbed

Our main server, hosting LuaGuardia, is based on a SGX-enabled Intel(R) Core(TM) i7-8700k CPU clocked at 3.7GHz with 32GB of RAM, running on Arch Linux with kernel version 5.4.23-1-lts. LuaGuardia is compiled in hardware/signed SGX mode, preventing any debugging or enclave monitoring. The client is running on a separate machine with same specifications. We also use a system based on an Intel(R) Core(TM) i7-6700 CPU with 16GB of RAM, acting as an HTTP server using machines are interconnected over a 1GbE wired network for the needs of `wrk2` application.

Evaluation Highlights Without any optimizations utilized, LuaGuardia introduces an average execution overhead of about 23% 6.2.1. Initialization optimizations can significantly increase LuaGuardia’s overall performance and specifically for short-lived Lua scripts 6.2.2. We can observe that system-call batching is responsible for 70% performance improvements 6.2.2. Finally, the overhead LuaGuardia introduced to large scale, real-world applications is an average of 17-19%.

6.1.1 Security Analysis

We validate and assess the security properties and guarantees offered by LuaGuardia by attempting and performing different attacks and demonstrating how our SGX-assisted system is able to mitigate them.

Data Integrity & Confidentiality A quite common technique for attacks is to exploit software vulnerabilities and bugs, either found on the target Lua script or on the same Lua interpreter, in order to inject malicious code. Such attacks target the control flow of the executing script within the SGX enclave, where quite often it aims to extract or tamper the sensitive data, code or dataset as shown in Figure 5.1. LuaGuardia encapsulates the entire Lua interpreter within the secure SGX enclaves; any Lua code that needs to be executed along with the corresponding data and modules, arrive at the server securely as mentioned in Sections 4.1.3, 4.1.2, in an encrypted format, where they are fetched and decrypted on-the-fly inside the enclave’s LuaGuardia virtual machine. Since all the required Lua code, input and dependencies are stored in the untrusted file-system, and their plain data reside inside the enclave only, our system is able to prevent access against fully privileged attackers.

Similarly to incoming data, the output logs of the executed Lua scripts are added with an extra layer of encryption using the shared AES key which was generated at previous phase. The encryption takes place inside the SGX enclave and the encrypted results are transmitted back to the client securely, in a cipher-text format. In order to verify our system’s integrity, we use `sgx-gdb` tool to attach to the LuaGuardia running enclave process, using administrative permissions (*i.e.*, root access); by attaching to the process we can attempt to tamper the execution in order to dump the executing code, the offloaded data and modules that are being processed. However, since we have compiled the enclave in `Release Mode`, all the debugging information as well as all the code symbols are stripped for the final executing binary, so the attacker may not extract any useful information by observing the code. Additionally, due to the implementation of the SGX hardware, actions such as altering data or injecting code are completely prohibited, thus always resulting in Segmentation Violations.

Controlling the Kernel In case of a full system compromise including the kernel, the execution of an unprotected Lua virtual machine, the file-system and the network interfaces may be monitored or manipulated. However, even if a malicious party can monitor and intercept the network traffic between LuaGuardia and its clients or the file-system, they are not able to obtain the client requests containing the code, the input data and the server executing responses, as both of them are encrypted throughout the entire path and are only decrypted inside the enclave. The same rules apply for the data that are stored persistently in the file-system as they are sealed before leaving the trusted environment and unsealed upon required. The unsealing operation takes place solely in the enclave and validates the persistently stored data before reuse. Moreover, even if the attacker manages to acquire full `read/write/execute` rights in the whole system or manipulate process execution via the kernel they still cannot tamper or monitor the code executing in the enclave due to its reverse-sandbox property. As a result, such attempts will result in Segmentation Violations since the enclave’s protected memory is not mappable outside of the enclave.

6.2 Micro Benchmarks

In this section, we provide extensive evaluation of LuaGuardia using a series of different micro benchmarks, that have as a goal to help us understand and explain the different variables that affect the overall performance of our system and implementation as well as how it performs in comparison to a vanilla Lua virtual machine. It is known that the execution performance of an SGX-enabled application is, almost, identical to its non-SGX variant as long as the code is pure, meaning it is self contained, has no external dependencies, does not perform system-calls or any I/O is invoked and all the required data are accessible in the enclave. Moreover, the I/O between the enclave and the untrusted world can be quite expensive as the execution has to be transferred between the enclave and its driving application and data are encrypted and decrypted each time they enter or leave the enclave as well as several additional checks are performed on the copied data for integrity and security purposes. An additional overhead is induced, which is linked with the storage and manipulation of data in the enclave, is the 128MB live protected memory limitation, discussed in Section 2.1.1. Storing and accessing data that exceed the live available protected memory shall trigger expensive page swapping that requires performing encryption of the page that will be swapped and the decryption of the decryption, each time this operation occurs.

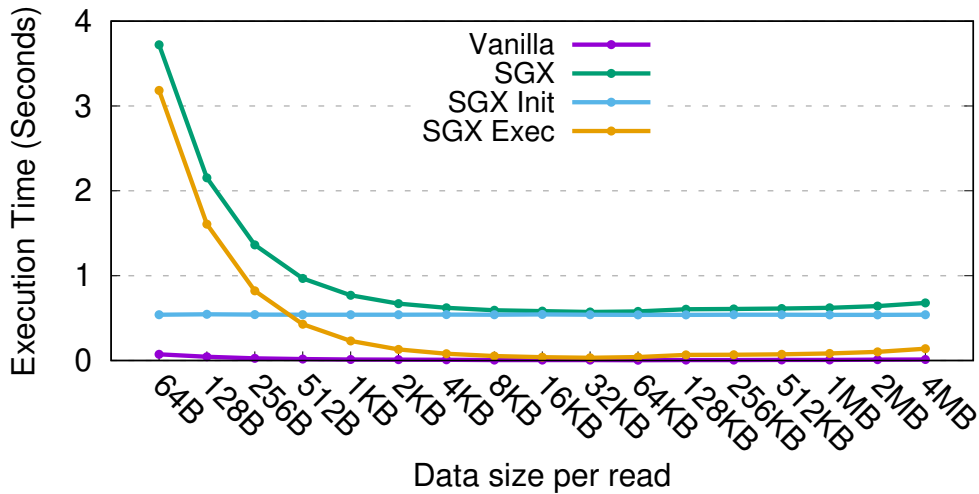


Figure 6.1: Performance comparison between the vanilla Lua virtual machine and LuaGuardia when reading a 32MB file with variable read buffer sizes

Data Retrieval In order to understand and evaluate the impact introduced by performing system-calls that are served via transferring the execution context to the untrusted part of the application as well as the overheads introduced by triggering the protected memory page swapping, we perform the following experiments. First, we develop a simple Lua benchmark script that reads data from a 32MB file from the untrusted file-system in

chunks ranging from 64B up to 4MB input size. We execute the script using both the original Lua virtual machine and LuaGuardia’s implementation and present the performance results in Figure 6.1, where `Vanilla` indicates the performance of the unmodified Lua virtual machine and `SGX` the end-to-end performance of LuaGuardia’s interpreter. The overall execution of LuaGuardia is broken down to `SGX Init` and `SGX Exec` indicating the time required to create and initialize the enclave entity of LuaGuardia and the time required only for the Lua script to complete code execution respectively. As we can see in the figure, increasing the read data buffer significantly reduces the execution time as fewer system-calls are involved and the execution time is transferred fewer times between the trusted and the unprotected spaces of LuaGuardia. Furthermore, the encryption process for the enclave inbound data is triggered less frequently. However, we notice that increasing the read data buffer beyond the input size of 2KB, has little to no effect in further increasing the performance. Additionally, the overall end-to-end execution of LuaGuardia is increased in comparison to the original Lua virtual machine due to the expensive enclave initialization while the required script execution time is, almost, identical to the stock interpreter, indicating that Intel SGX does not introduce significant performance overheads when carefully designing the enclave I/O channels and requirements.

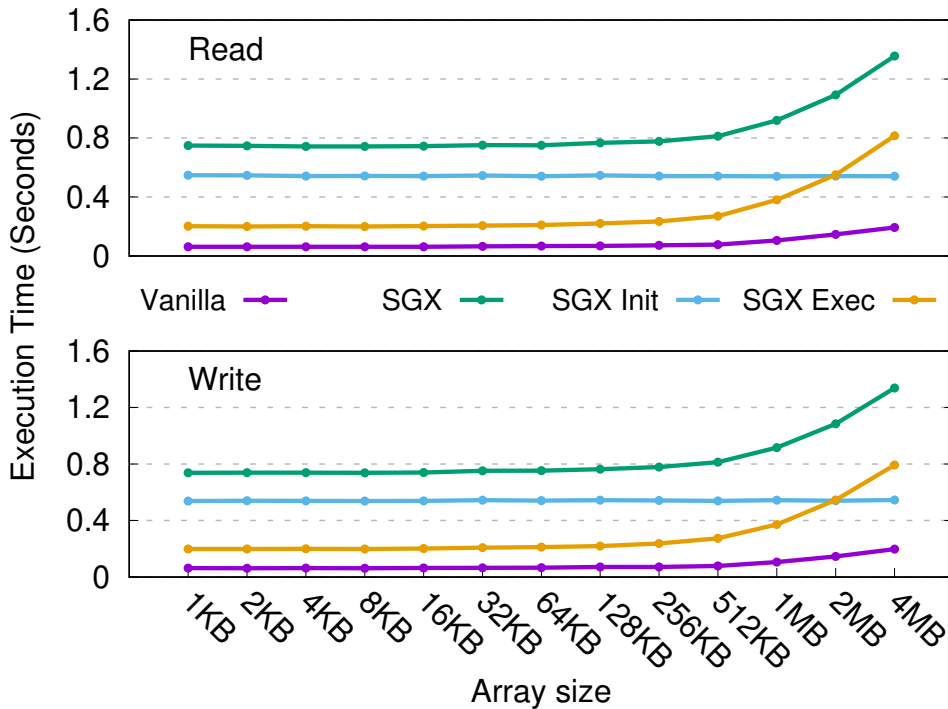


Figure 6.2: Performance comparison between the vanilla Lua virtual machine and LuaGuardia when randomly performing one million accesses (read/write) to memory locations ranging between 1KB and 4MB

Memory Accesses The next step to our evaluation process is to measure the performance overhead introduced by randomly accessing the enclave protected memory with and without triggering page swapping. In order to achieve such result, we design a simple Lua benchmark script that performs 1 million random memory accesses to consecutive protected memory spaces, ranging from 1KB to 4MB. We execute the script in two different ways, the first time we perform random 1B writes while the second 1B reads at each random access. The results of this analysis are presented in Figure 6.2. We notice that the memory access times achieved by LuaGuardia, are almost identical to those achieved by the original Lua Virtual Machine when enclave page swapping is not triggered and the target memory space resides in the live protected memory area. Moreover, we plan the allocation of consecutive memory spaces exceeding 1MB so that we start triggering the memory page swapping mechanism offered by Linux. In such cases, we can observe that the page swapping effects affect the overall execution of the system, enforcing a linear performance degradation as the amount of non-live protected memory increases.

6.2.1 Benchmark Applications

Cryptographic Benchmarks Our main goal here is to observe and understand the properties and parameters that affect LuaGuardia’s performance. To do so, we evaluate and execute 12 popular cryptographic algorithms using the original Lua virtual machine and LuaGuardia’s interpreter, recording only the script execution time for both of the methods. The selected cryptographic algorithms provide a representative combination of heavy arithmetic operations along with memory resource utilization, which can stress the systems under test, in terms of excessive computations. The results of this experiment are presented in Figure 6.3. The execution time is normalized to indicate the processing of 1MB input of each one of the cryptographic algorithms. Furthermore, we present the performance overhead introduced on top of each bar cluster, roundup to 0.5. The results indicate that the average performance overhead introduced by LuaGuardia may reach up to 23% whereas SHA256 yields the highest performance overhead, reaching up to 46%. This behaviour reported for SHA256 is mainly attributed to the increased memory requirements of this implementation which triggers protected memory page swapping.

Benchmark Suite In this section, we evaluate and profile LuaGuardia using 12 popular benchmark applications, developed in pure Lua. The purpose of this evaluation is to understand LuaGuardia’s performance properties when utilizing its two different execution methods, (i) local and (ii) remote execution. As described in Section 4.2, when LuaGuardia operates in `local` execution mode, both the target Lua script(s) and their data reside on the same physical machine, in contrast with `remote` execution mode, where the Lua scripts, the code modules and the additional environment dependencies are securely transmitted by the client, over the network.

Table 6.1 shows a short description of each of the benchmarks chosen for this analysis. Overall, they cover a wide range of different properties, including computational, memory, and I/O-intensive workloads. By doing so, we are able to exercise different characteristics of LuaGuardia and compare it against the vanilla Lua virtual machine.

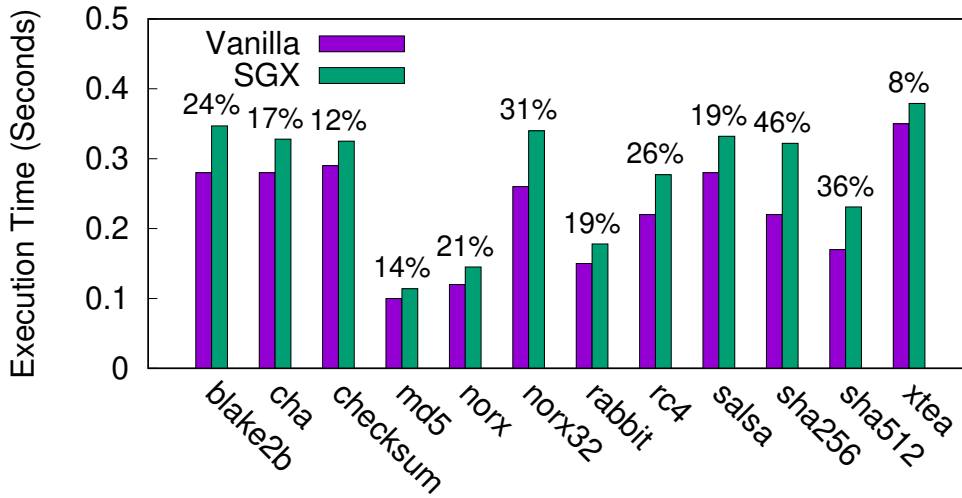


Figure 6.3: Performance sustained for cryptographic benchmarks

We execute each benchmark ten times using the vanilla Lua virtual machine and the two operation modes provided by LuaGuardia and report the average end-to-end execution time in Figure 6.4. The reported values for LuaGuardia contain the time required to initialize the server, including the initialization of the SGX enclaves and the original Lua interpreter, as well as the encryption and decryption operations required in order to secure the incoming Lua scripts, their input data and modules, as well as the results. Moreover, in the case of remote execution, the overall time also includes the network I/O.

As we can see in the figure, the stock Lua interpreter yields significantly better results compared to LuaGuardia when benchmarks with very low computational needs are executed. However, the delta between the stock Lua virtual machine and LuaGuardia decreases as the benchmarks become more computational intensive or demand more I/O and thus require more time to execute. Also, the performance overhead introduced by the network I/O when LuaGuardia is executed remotely is quite minimal as the encryption and decryption of the incoming data and each script’s output is performed in both methods and is not offloaded to the network layer. These results indicate that major overhead introduced by LuaGuardia mostly derives by its expensive initialization, as between executions the server is always terminated and re-initialized.

6.2.2 Performance Optimizations

As described in the previous section, the initialization time of the SGX enclaves and the Lua virtual machine introduce a significant performance overhead, especially when executing lightweight scripts that are not computation-intensive, yielding sub-second performance. To further understand the various overheads introduced by LuaGuardia’s software stack, we re-evaluate LuaGuardia with the same benchmarks, reposing the execution time breakdown into three different categories: (i) network I/O, (ii) initialization and (iii) execution. The values reported for initialization contain the time required for initializing

Table 6.1: Benchmark Operations

Micro benchmark	Operation
deltablue	Object-oriented constraint solver
life	Game of Life puzzle
mandelbrot	Mandelbrot computation
queens	Solves the Queens puzzle
coll.detect	Airplane collision detection simulation
fasta	Generates DNA sequences
ray	Ray casting simulation
richards	Operating system kernel simulation
bin.trees	Creates perfect binary trees
havlak	Loop recognition algorithm
nbody	n-body simulation of solar system
recurs.fib	Performs recursive Fibonacci

both the enclaves and the protected Lua virtual machine, while the execution time corresponds to the required cryptographic operations. As shown in Figure 6.5, we notice that the initialization time overshadows the execution of the faster benchmarks when the initialization optimizations are disabled. This overhead can be further exaggerated in cases where clients need to repeatedly offload the execution of lightweight, self-contained, functions that yield very small execution times (e.g., sub-second).

Initialization Optimizations In order to lift this limitation, we perform the following optimization. To overcome these start-up overheads, we re-design LuaGuardia to use pre-initialized enclaves. In particular, the SGX enclave that hosts the protected Lua virtual machine, as well as its driving code is initialized only once during bootstrap and remains active throughout the entire server’s lifetime. Afterwards, each client request yields the initialization of the protected Lua virtual machine, each time reusing the always-active SGX enclave. The Lua virtual machine is always re-initialized upon each client request, discarding the previous Lua state, to ensure the confidentiality of previous executions.

Once the initialization optimization is in place, we evaluate LuaGuardia with the same benchmark setup, reporting again the execution time breakdown. The results of this analysis are depicted in the bottom row of Figure 6.5. Comparing the overall execution time breakdown of the new LuaGuardia version we notice that the optimized design significantly reduces the initialization time allowing for faster end-to-end execution, especially for short-lived Lua scripts.

system-call Batching In order to evaluate our second optimization, the system-call batching, we design and implement a Lua benchmark that iteratively performs a system-call, in this case `write()`. We execute the script multiple times, each time increasing the number of requested system-calls, ranging between 10 thousand up to one million, using the vanilla Lua virtual machine and LuaGuardia with and without system call batching.

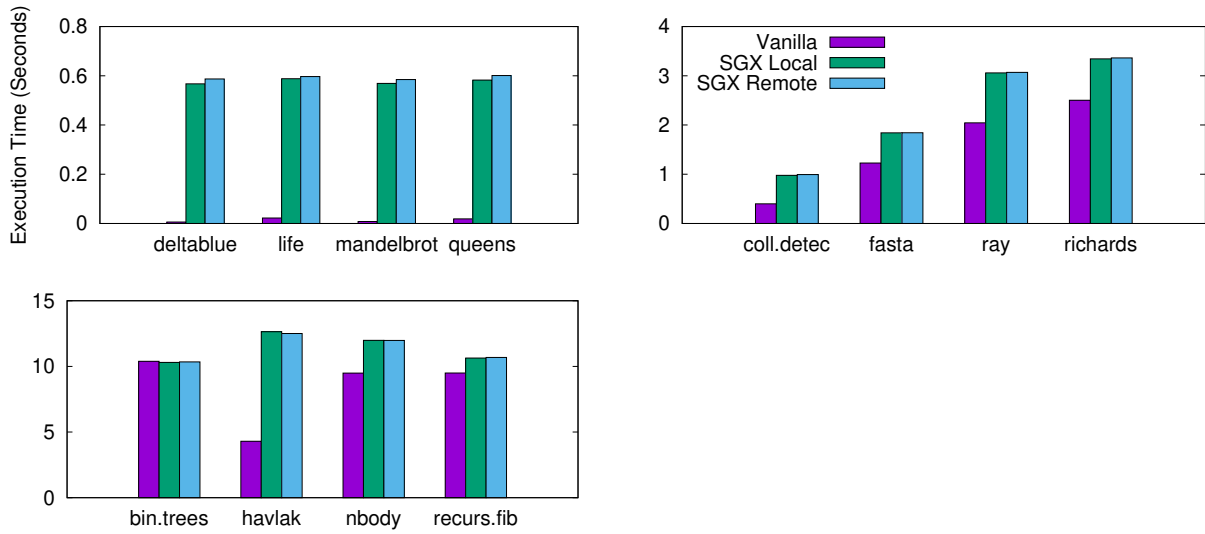


Figure 6.4: End-to-end performance when executing 12 popular Lua benchmarks.

When system-call batching is enabled, LuaGuardia hooks the function and stores the results to be written in a local buffer, that resides in the enclave, instead of performing an `SGX OCALL` each time. At the end of the execution, the results are written to the script’s desired location using a single `OCALL`. As displayed in Figure 6.6, we notice that the overall execution performance is increased by an average of 70% for applications performing multiple system-calls.

Optimization Results We now re-evaluate LuaGuardia with all optimizations enabled, using the same experimental setup that has been described in Section 6.2.1. Figure 6.7, compares the execution time achieved by the stock Lua virtual machine with the un-optimized and optimised LuaGuardia implementations. The outcome of this evaluation clearly indicates that the aforementioned optimization increased the performance capabilities of LuaGuardia and significantly decreased the delta between the vanilla Lua virtual machine and our system when executing short-lived Lua scripts. Moreover, for more computationally intensive scripts, the performance achieved by LuaGuardia is almost comparable to the stock Lua virtual machine, with the only exception being `havlak` due to its high live memory requirements which constantly trigger enclave memory page swapping.

6.2.3 Real World Application 1: wrk2

In this section we evaluate `wrk2`’s performance when executing over LuaGuardia. `wrk2` [53] is a modified version of the original `wrk` HTTP benchmarking tool, which uses Lua scripts for the generation of HTTP requests, the processing of the responses and any kind of reporting. At its original implementation, `wrk2` is based upon LuaJIT. However, the provided scripts are also executable via the stock Lua virtual machine, and consecutively

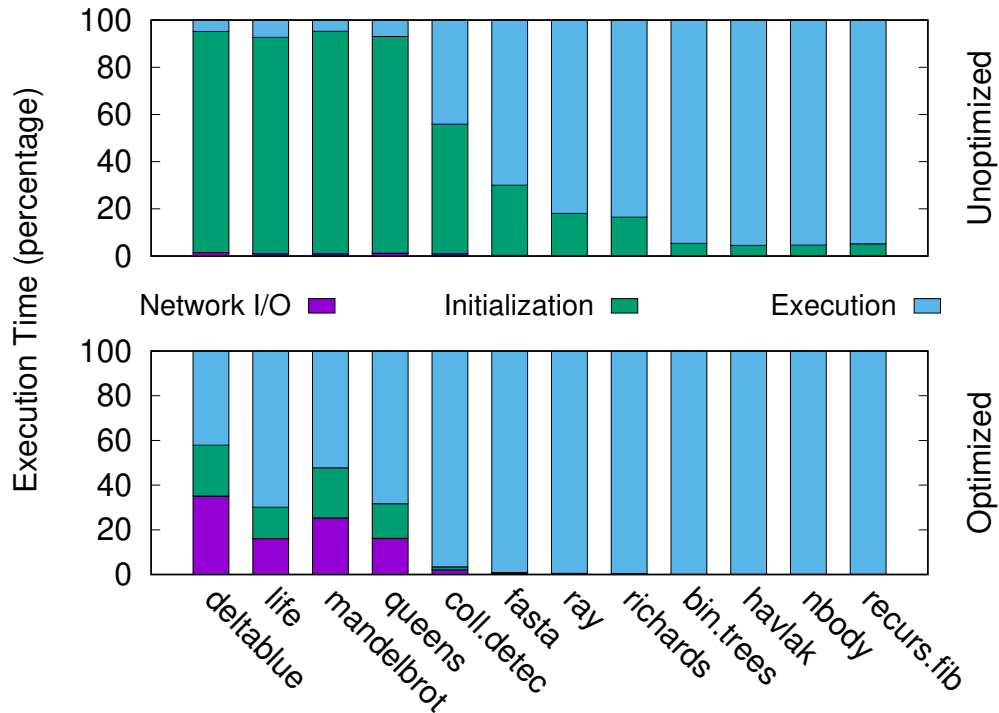


Figure 6.5: LuaGuardia execution time breakdown with and without server initialization optimizations.

by LuaGuardia. We choose to evaluate our system using wrk2 since it serves as a good example of networking applications that generates synthetic load and requires external services, such as an HTTP server.

Experimental Setup Besides the two machines for the LuaGuardia server and client, we also setup an external HTTP server, running `darkhttpd`, which wrk2 will connect to and perform the benchmarking operations. For our analysis, we execute three different wrk2 Lua scripts, (i) `Auth`, (ii) `Counter` and (iii) `Report`. The first script performs response handling and retrieves an authentication token. The second script changes the request path and header for each request while the third Lua script implements a custom `done()` method that reports latency percentiles in a CSV format.

Evaluation Process We use the LuaGuardia client to execute wrk2 which connects to the HTTP server and each time initiates one of the three different Lua scripts. For each HTTP request, wrk2 executes the target script by offloading it to the LuaGuardia server. Upon script execution, the client receives the script's output and finalizes the operation. This is a particularly different design compared to the other applications since each script is re-executed upon each HTTP requests. This means that for each packet transmitted to the HTTP server, LuaGuardia's protected Lua virtual machine is reinitialized in order

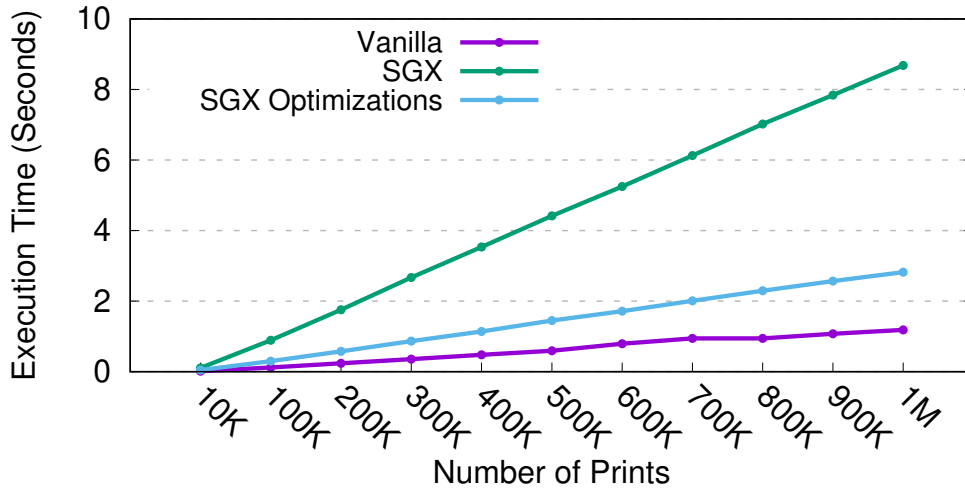


Figure 6.6: Performance comparison between the vanilla Lua virtual machine, LuaGuardia and LuaGuardia with system-call batching enabled.

to execute the appropriate wrk2 Lua script. The design followed by wrk2 serves as a good example of clients constantly requesting script executions with minimal data and low computation requirements and is ideal in order to stress LuaGuardia’s initialization optimizations.

Results The results obtained after executing the three wrk2 Lua scripts using both the stock Lua virtual machine and LuaGuardia are presented in Figure 6.8. The X-axis indicates the executed scripts while the Y1-axis indicates the sustainable throughput in KB/sec. The bar clusters follow the Y1-axis and the throughput achieved by the stock Lua virtual machine is marked as `Vanilla xput` while the values reported by LuaGuardia are marked as `SGX xput`. The Y2-axis indicates the average requests per second while the achievable values are presented with lines/points, following a similar marking. The overhead introduced by LuaGuardia is also reported on top of each bar cluster with each value being round up to 0.5. Observing the results, we notice that LuaGuardia introduces 34% overhead for the `Auth` benchmark and 28% overhead for the `Counter` benchmark. The main reason behind this behaviour is that `Auth` triggers an authentication token retrieval that stresses the I/O path more than the other three benchmarks. On the other hand, we can see that `Report` yields almost the same throughput and requests/sec result since the script is only executed once and requires very minimal processing and I/O.

6.2.4 Real World Application 2: Snabb/pflua

The second real-world application that we choose to evaluate LuaGuardia with is Snabb [55]. Snabb is a virtualised Ethernet toolkit that allows the implementation of networking applications using Lua. Originally, Snabb utilizes the LuaJIT instead of the Lua virtual

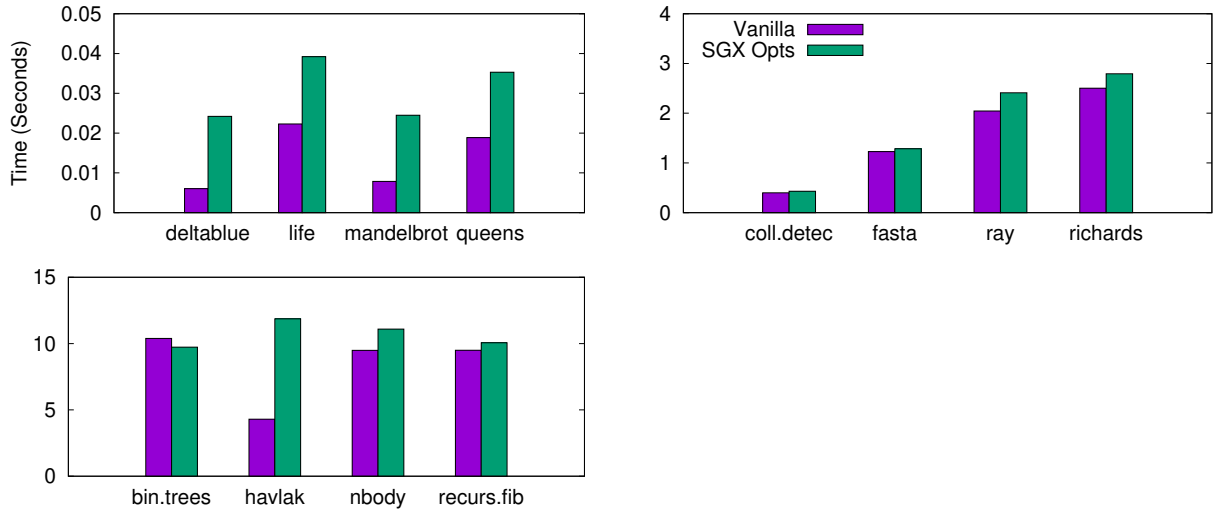


Figure 6.7: Performance sustained when applying system-call batching and enclave pre-initialization.

machine, however, the developed scripts are also able to be executed with the stock Lua virtual machine as well as with LuaGuardia. One of the many applications provided by Snabb is a packet filtering module, namely `pflua`. The `pflua` packet matcher is developed with Lua, using 9600K LoC, and filters incoming packets based on `pflang` [56], a filter that is also used by `tcpdump` [57].

Dataset In order to execute and evaluate Snabb/pflua, we use a dataset of 12 pcap traces, each one containing 32 thousand packets. One of the traces, labeled `uni`, is a properly de-anonymized and GDPR compliant real traffic trace, collected at an educational institute. The rest of the traces are provided by Snabb as test cases, each one divided by its protocol. In order to properly evaluate LuaGuardia with these traces, we expand Snabb’s synthetic traces by replaying them in order to increase their size to 32 thousand packets. The rule-set provided to `pflua` contains one hundred custom rules, defined in `pflang`, and is used in order to process all pcap traces. We design the rule-set in such a way that multiple rules are being triggered by each trace, making this way the workload uniform and comparable across all input data.

Evaluation Process Once all the required data and the rule-set are gathered, we proceed with the evaluation of Snabb/pflua in the following way. First, we execute Snabb in order to extract the packet information from each pcap trace and transform it to a `pflua`-compatible format. Internally, Snabb uses L7 firewall which at each iteration extracts the packet data, transforms it and then forwards the output to LuaGuardia that executes the `pflua` Lua code. The rule-set utilized by `pflua` also resides in LuaGuardia’s SGX enclave. In this way, it is not readable or modifiable by malicious parties that wish to monitor the

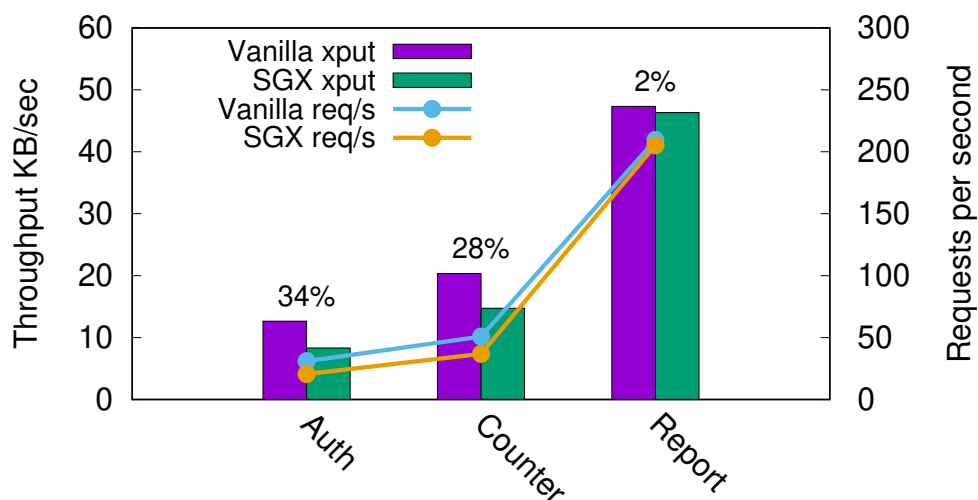


Figure 6.8: Performance sustained for wrk2

packet filtering process or alter the applied rules. Once each packet reaches the SGX enclave, it is being processed by the pflua module which buffers the results and returns them back to the Snabb client when the entire trace is processed. We repeat this operation for every trace, using both the vanilla Lua virtual machine and LuaGuardia.

Results The results of this analysis are presented in Figure 6.9. The X-axis indicates the pcap trace being filtered while the Y-axis indicates the overall execution time. The end-to-end execution time achieved by the stock Lua virtual machine is tagged as *Vanilla* while *SGX* reports the execution time required by the LuaGuardia server with all optimizations enabled. On top of each bar cluster we report the overhead introduced by LuaGuardia, round up to 0.5. As we can see in the figure, LuaGuardia introduces less than 20% overhead for most of the traces while the average overhead is 19%. The biggest delta is reported when processing the *BitTorrent* pcap trace, reaching 41%. We attribute this overhead to the design of pflua’s filtering for the following reason. Each packet in the trace contains a significantly big payload that has to be transferred inside LuaGuardia’s SGX enclave, despite the fact that our filtering only applies rules based on packet headers. For this reason, a considerable amount of data has to be encrypted, transferred and decrypted without actually being required by the packet filtering process. Moreover, the processing required to filter each packet based on its header is quite minimal, thus being unable to hide such overhead. In contrast, the stock Lua virtual machine does not need to perform this costly transfer operation. However, we can see that LuaGuardia is able to process the real-world traffic trace, namely *uni*, with a minimal overhead of 14%.

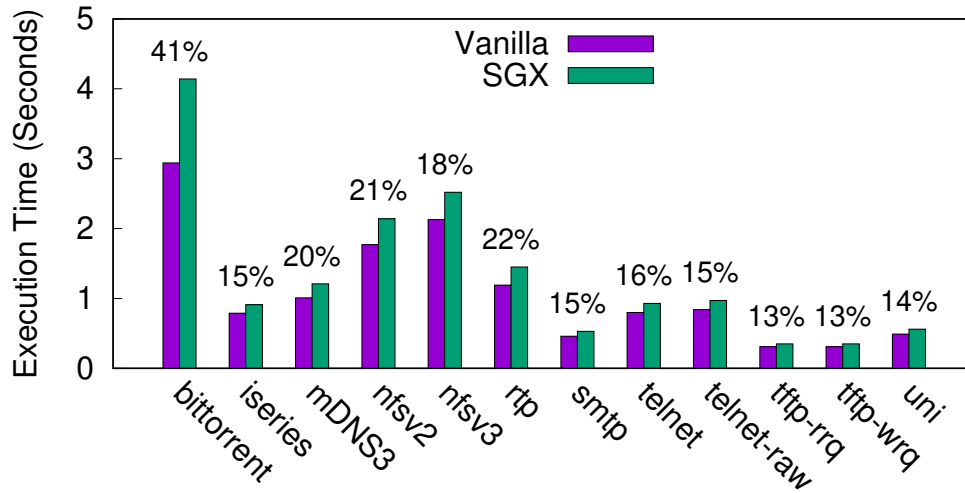


Figure 6.9: Performance sustained for pflua

6.2.5 Real World Application 3: Snabb/IPsec

As described in the previous section, Snabb is a simple Ethernet toolkit that enables the development of networking applications using Lua. For our third real-world application, we design and implement a custom module for Snabb. The module utilizes Snabb in order to decode pcap traces and transform them into JSON. Afterwards, the transformed packets are forwarded to our Lua module, executed by LuaGuardia, which is responsible for encrypting and decrypting each packet, operating as a lite IPsec module.

Data We evaluate our custom module using the pcap traces described in Section 6.2.4. Since our custom module is based on Snabb, similar to the previously discussed Snabb/pflua, we choose not to modify our pcap dataset for consistency. Moreover, in this way we will be able to observe and compare the overhead introduced by LuaGuardia when operating on the same data but performing a different type of computation. More specifically, we aim to observe if the performance delta between the stock Lua virtual machine and LuaGuardia is decreased when a computational intensive task takes place alongside an I/O intensive operation such as transferring network traffic to and from the SGX enclaves.

Evaluation Process We begin our Snabb/IPsec module development by implementing a configuration that specifies the functions responsible for parsing and processing pcap traces. More specifically, we replace the I/O facility provided by Snabb with our own implementation which exposes an `SGXreader` and an `SGXwriter` object. Upon initialization, we create a new `SGXreader` and `SGXwriter` context. Afterwards, using the `pull` method, exposed by the `SGXreader`, we iterate each packet found in a pcap file and encode it using JSON. Each encoded packet is then written to the output using the `push()` method provided by the `SGXwriter`. Once the entire trace is processed, it is forwarded to our IPsec module, executed securely by the LuaGuardia server.

Internally, the module receives the JSON encoded packets and performs encryption and decryption of each packet payload. This operation is performed using ChaCha20-Poly1305. Once every packet in the pcap trace passes a round of encryption and decryption, the results are forwarded back to the client. After carefully inspecting the module’s correctness, we execute Snabb/IPsec using both the vanilla Lua virtual machine as well as LuaGuardia with every optimization enabled, each time processing a different pcap trace.

Results The evaluation results obtained after the module’s execution are presented in Figure 6.10. In a similar fashion to the previous analysis, the X-axis indicates the pcap trace being processed while the Y-axis indicates the overall execution time. We mark the end-to-end execution time achieved by the vanilla Lua virtual machine as `Vanilla` and the results obtained by the execution of the LuaGuardia server as `SGX`. On top of each bar cluster we report the overhead introduced by LuaGuardia, round up to 0.5. The evaluation indicates that the processing of most of the traces using LuaGuardia, introduces a 15% performance overhead. The average overhead is 17%, 2% lower than the value reported for Snabb/pflua. This observation, along with the fact that the execution times are one order of magnitude bigger than those reported for Snabb/pflua, is an initial indication of our expectation — computational-intensive tasks tend to overshadow the I/O overhead introduced by SGX enclaves. This is more evident when comparing the overhead introduced by LuaGuardia when processing the `bittorrent` pcap trace using Snabb/pflua and Snabb/IPsec. We can see that the overhead is reduced by 18% since the high execution time required to encrypt and decrypt each packet in the trace tends to overshadow the time needed to transfer the big packet payloads into the enclave.

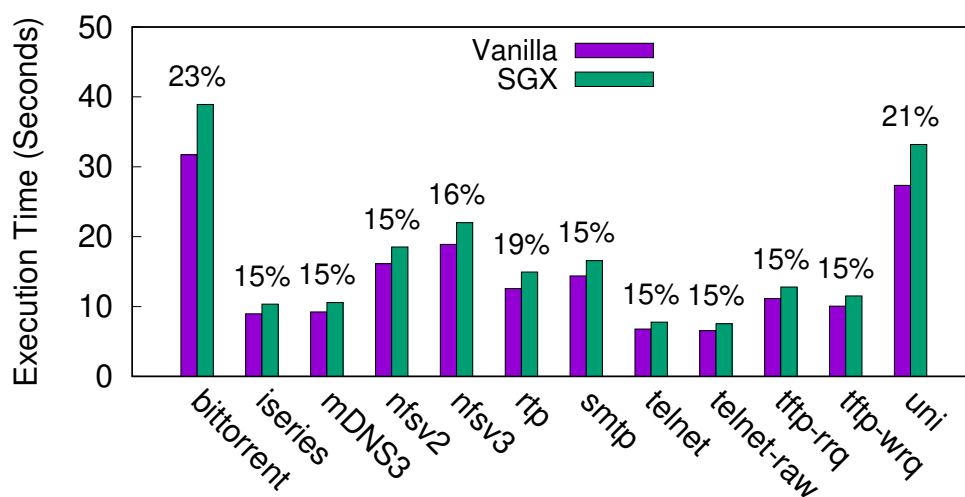


Figure 6.10: Performance sustained for IPsec

Chapter 7

Discussion And Limitations

7.1 Local Execution Mode

As mentioned in Section 4.2, even though Local Execution mode is supported by LuaGuardia, we opt out not to use it, for the already mentioned reasons. As a result, even the same machine that runs the LuaGuardia server, may not use and leverage the SGX capabilities for its own use, since a malicious attacker may tamper the execution scripts due to the takeover of the file-system and the I/O. One solution to this issue, assuming that the attacker has full take-over of the system including the fs, the system kernel and the peripherals, would be to preload the different Lua scripts with their data in the LuaGuardia enclave and perform code requests to the enclave. However, this solution disables LuaGuardia's ability to execute code dynamically, thus we opt not to use it.

7.2 Native Module Support

As mentioned in Section 4.1.1, the `dlopen` family of functions is not supported nor its use is endorsed within the enclave. This occurs due to the fact that `dlopen` is a system-call, and thus has to be served and executed in the untrusted part of the application. As a result, the memory mapping of the requested shared library will be performed and allocated at non secure memory, thus it will introduce several security risks, since LuaGuardia's enclave cannot monitor nor validate the integrity of the untrusted memory. A solution to this issue would be to pre-compile and include the required group of native Lua libraries in the main LuaGuardia interpreter. This would allow and help LuaGuardia virtual machine to have better compatibility with the legacy Lua scripts, but would introduce several problems. First of all, including and compiling additional to the enclave would increase the TCB size of the enclave, that might trigger the memory page swapping in case the live used memory of the enclave exceeds the 128MB, resulting in decreased performance. Secondly, adding third-party code to the enclave that is not trusted, might contain memory leaks or bugs that may risk LuaGuardia's environment and facilitate new kind of attacks. Finally, adding new code to the enclave requires that the developer has SGX insight and understands the structure of the project. Of course, this solution is intended only for SGX

developers that have deep understanding of the project architecture and security primitives.

7.3 Enclave Size

Another problem regarding application development using Intel SGX concerns the memory limitations and constraints introduced by the underlying hardware. As mentioned in Section 2.1.1, the limitations of the available live memory is limited to 128MB totally in the system. Due to this issue, several problems arise. First of all, all the enclaves secure memory allocations have access only to this 128MB memory, whereas any request that exceeds this limitation, triggers the swapping mechanism in the system, resulting into slower performance of all the applications that leverage and utilize enclave functionality. In LuaGuardia, we acknowledge this issue and we opted to use 150MB of live memory. This choice was made since it was the bare minimum required in order to perform all the mentioned micro and macro benchmarks in the previous section. Furthermore, keeping the TCB small is an aspect that all the application developers should keep in mind when developing in TEEs platforms. Even though LuaGuardia is fully capable of executing Lua scripts and may work as a drop-in replacement of Lua5.3, executing whole applications within the enclave is not endorsed. However, the option to increase the max available memory(using swapping) may be increased by LuaGuardia's configuration.

Chapter 8

Related Work

VC3 [58] uses SGX to achieve confidentiality and integrity for the Map Reduce framework. Haven [59] aims to execute unmodified legacy Windows applications inside SGX enclaves by porting a Windows libOS into SGX. Similarly, Graphene-SGX [60] encapsulates the entire libOS, including the unmodified application binary, supporting libraries, and a trusted runtime with a customized C library and ELF loader inside an SGX enclave. SCONE [61] is a shielded execution framework that enables developers to compile their C applications into Docker containers. These works are either domain-specific or provide a container-based or libOS approach in which users can run general-purpose applications. LuaGuardia provides a balance between these approaches by providing general purpose program execution through a high-level programming language, such as Lua, that also keeps the TCB size minimal.

Ryoan [62] provides a distributed framework that utilizes hardware SGX enclaves to protect sandbox instances, written in C, from potentially malicious computing platforms. It is designed as such confined modules operate on the given input once and do not hold their state (similar to LuaGuardia) preventing potential data and state leakage. Glamdring [63] is a source-level partitioning framework that secures applications written in plain C. The developer has explicitly to pinpoint the sensitive and crucial data using annotations. Then Glamdring automatically identifies and partitions the code into untrusted and enclave code parts. In contrast with LuaGuardia, the developer has to recompile the code with the newly added annotations and then perform the analysis to partition the code whereas on LuaGuardia neither additional annotations are needed nor any code analysis.

Similar to our work, Civet [4] is a framework developed concurrently with LuaGuardia and partitions Java applications into the SGX enclaves. The framework also provides garbage collection but introduces significantly greater overhead compared to our system. TrustJS [64] also explores the possibility of bridging Javascript language with SGX enclaves in securing security-sensitive Javascript components inside browser applications. However, TrustJS requires script partitioning regarding the trusted scripts, data and meta information. Additionally, it does not attempt to provide a general execution sandbox for legacy Javascript applications. ScriptShield [65] enables development in SGX enclaves using high level scripting languages such as Lua, JS, Squirrel. In contrast

to LuaGuardia though, it fails to demonstrate its practicality on real-life applications, in terms of performance. On LuaGuardia, we have performed all the necessary optimizations that makes it practical to use in real-life applications. [66] introduces a reactive middleware framework approach for data stream processing on the cloud based on Intel SGX and Lua. Overall, LuaGuardia builds on the same objectives, however provides many optimizations that are necessary to make it practical in terms of performance.

Finally, several improvements for SGX have been recently developed, such as SGXBOUNDS [67], SGX-Shield [68], Eleos [69] and T-SGX [70]. All these works are orthogonal to our approach and can be integrated to LuaGuardia.

Chapter 9

Conclusions and Future Work

In this section we present a summary of the contributions of this work (§ 9.1) and some thoughts on potential future work (§ 9.2).

9.1 Summary of Contributions

In this work, we develop a novel dynamic code execution framework that supports code offloading and execution of Lua legacy applications with their respective code modules and required environment dependencies. More specifically, the ported virtual machine is able to achieve and have close performance to the original Lua interpreter, in case where the executing code has no external dependencies after virtual machine and code bootstrap phase. In addition, our framework functionality may be enriched by adding additional Lua modules. Furthermore, we hook the underlying system-call dependencies of Lua virtual machine, intercept them and offload them to the untrusted OS for execution, without affecting the normal script execution. In this way, we achieve a complete mediation of the OS-enclave interface.

9.2 Future Work

As future work, we plan on exploring more high level languages similar to Lua such as Javascript, Java and Python on the same concept but for different purposes, in which every language is strong at. Additionally, the same concept of LuaGuardia can also be applied on ARM TrustZone hardware. Arm's TEE offers an alternative approach that its main targets are mobile phones, IoT devices and smart devices. As a result, we could apply and enforce LuaGuardia or similar approaches based on ARM TrustZone, that leverage high level language code execution to strengthen the security of Android devices or its whole ecosystem of devices. Another interesting aspect for improvement for LuaGuardia and similar projects would be to dynamically load and execute shared libraries alongside with the Lua or any other high level language scripts inside the trusted execution environment. By doing so, we can almost achieve perfect Lua code compatibility.

9.3 Conclusion

LuaGuardia allows the easy development of confidential computing through a lightweight, yet extensible, programming language, such as Lua. By doing so, it remedies the complexity of application signing, as well as the execution of dynamically-loaded code. LuaGuardia enables several optimizations at the interpreter level which allow protected code execution with reasonable performance overheads. Our evaluation results show that it can be used by a diversified set of applications, with reasonable overheads.

The evaluation of our system shows normal performance capabilities without adding a huge slowdown to the executing code. The performance of our implementation averages 13–41% (avg: 18%). Applying several optimizations lowers the overheads to 5–95%, especially for short-lived program fragments.

Bibliography

- [1] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede, “Hardware-based trusted computing architectures for isolation and attestation,” *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 361–374, 2017.
- [2] “ARM Security Technology, Building a Secure System using TrustZone Technology,” http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.
- [3] “Intel Software Guard Extensions (Intel SGX),” <https://software.intel.com/en-us/sgx>.
- [4] C.-C. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, “Civet: An efficient java partitioning framework for hardware enclaves,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.
- [5] “Open Enclave SDK,” <https://openenclave.io/sdk/>.
- [6] “Asylo: An open and flexible framework for enclave applications,” <https://asylo.dev/>.
- [7] M. S. Melara, M. J. Freedman, and M. Bowman, “Enclavedom: Privilege separation for large-tcb applications in trusted execution environments,” *arXiv preprint arXiv:1907.13245*, 2019.
- [8] “Intel SGX Attestation,” <https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/attestation-services.html>.
- [9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: A virtual machine-based platform for trusted computing,” in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 193–206.
- [10] “ADVANCED TRUSTED ENVIRONMENT: OMTP TR1,” http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf.
- [11] “TEE System Architecture,” http://read.pudn.com/downloads788/ebook/3112995/GP/GPD_TEE_SystemArch_v1.0.pdf.

- [12] A. Vasudevan, J. M. McCune, and J. Newsome, *Trustworthy execution on mobile devices*. Springer, 2014.
- [13] M. Sabt, M. Achemlal, and A. Bouabdallah, “Trusted execution environment: what it is, and what it is not,” in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 57–64.
- [14] “Introduction to Trusted Execution Environment: ARM’s TrustZone,” <https://blog.quarkslab.com/introduction-to-trusted-execution-environment-arms-trustzone.html>.
- [15] “What is a Trusted Execution Environment (TEE)?” <https://www.trustonic.com/news/technology/what-is-a-trusted-execution-environment-tee/>.
- [16] N. Asokan, J.-E. Ekberg, K. Kostianen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, “Mobile trusted computing,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1189–1206, 2014.
- [17] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, “Smart: Secure and minimal architecture for (establishing dynamic) root of trust.” in *Ndss*, vol. 12, 2012, pp. 1–15.
- [18] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, “Intel® software guard extensions: Epid provisioning and attestation services,” *White Paper*, vol. 1, no. 1-10, p. 119, 2016.
- [19] “Introduction to Trusted Execution Environments,” <https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf>.
- [20] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, “Secret: Secure channel between rich execution environment and trusted execution environment.” in *NDSS*, 2015.
- [21] V. Costan and S. Devadas, “Intel sgx explained.” *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.
- [22] “Intel Software Guard Extensions Developer Reference for Linux OS,” https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_Developer_Reference_Linux_2.10_Open_Source.pdf.
- [23] S. Pinto and N. Santos, “Demystifying arm trustzone: A comprehensive survey,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.
- [24] “MOBILE (ANDROID) HARDWARE STATS 2017,” <https://web.archive.org/web/20170808222202/http://hwstats.unity3d.com:80/mobile/cpu-android.html>.
- [25] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stempf, “Sanctuary: Arming trustzone with user-space enclaves.” in *NDSS*, 2019.

- [26] “Secure Partition Manager,” https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/include/spm.html.
- [27] “ARM Cortex-A57 MPCore Processor Technical Reference Manual,” <https://developer.arm.com/documentation/ddi0488/d/programmers-model/armv8-architecture-concepts/exception-levels>.
- [28] “OpTEE,” <https://www.op-tee.org/>.
- [29] “Nvidia Trusty TEE,” <https://docs.nvidia.com/jetson/l4t/index.html>.
- [30] “Trustonic,” <https://www.trustonic.com/>.
- [31] “Google Trusty TEE,” <https://source.android.com/security/trusty>.
- [32] “Samsung TEEGRIS,” <https://developer.samsung.com/teegris/overview.html>.
- [33] J. Amacher and V. Schiavoni, “On the performance of arm trustzone,” in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2019, pp. 133–151.
- [34] “Arm TrustZone Technology,” <https://developer.arm.com/ip-products/security-ip/trustzone>.
- [35] S. Zhao, Q. Zhang, G. Hu, Y. Qin, and D. Feng, “Providing root of trust for arm trustzone using on-chip sram,” in *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, 2014, pp. 25–36.
- [36] “Samsung Knox Developer Documentation,” <https://docs.samsungknox.com/dev/knox-attestation/about-attestation.htm>.
- [37] “Lua Language,” <http://www.lua.org/about.html>.
- [38] “Adobe Lightroom,” <https://www.adobe.io/apis/creativecloud/lightroomclassic.html>.
- [39] “Wireshark,” <https://www.wireshark.org/>.
- [40] “Snort,” <https://www.snort.org/>.
- [41] “World of Warcraft,” <https://wowwiki.fandom.com/wiki/Lua>.
- [42] “Lua Unofficial FAQ (uFAQ),” <https://www.luafaq.org/>.
- [43] “Lua 5.3 Reference Manual,” <https://www.lua.org/manual/5.3/>.
- [44] “Lua Technical Notes,” <https://www.lua.org/notes/ltn002.html>.
- [45] “Lua Environment and Modules,” https://www.lua.org/manual/2.1/section3_2.html.

- [46] L. H. De Figueiredo, R. Ierusalimschy, and W. Celes Filho, “Lua: an extensible embedded language,” <https://www.drdobbs.com/open-source/lua-an-extensible-embedded-language/184410014>.
- [47] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, “Malware guard extension: Using sgx to conceal cache attacks,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.
- [48] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiaainen, S. Capkun, and A.-R. Sadeghi, “Software grand exposure: {SGX} cache attacks are practical,” in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.
- [49] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, “Cache attacks on intel sgx,” in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.
- [50] E. Rescorla, “Rfc2631: Diffie-hellman key agreement method,” USA, 1999.
- [51] M. Pall, “The luajit project,” *Web site: <http://luajit.org>*, vol. 1015, 2008.
- [52] B. C. Xing, M. Shanahan, and R. Leslie-Hurd, “Intel® software guard extensions (intel® sgx) software support for dynamic memory allocation inside an enclave,” in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, 2016, pp. 1–9.
- [53] “A constant throughput, correct latency recording variant of wrk,” <https://github.com/giltene/wrk2>.
- [54] “Packet filtering in Lua,” <https://github.com/Igalia/pflua>.
- [55] “Snabb: Simple and fast packet networking,” <https://github.com/snabbco/snabb>.
- [56] “PFlang,” <https://github.com/Igalia/pflua/blob/master/doc/pflang.md>.
- [57] “tcpdump,” <https://www.tcpdump.org/manpages/tcpdump.1.html>.
- [58] 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 *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 38–54.
- [59] A. Baumann, M. Peinado, and G. Hunt, “Shielding applications from an untrusted cloud with haven,” *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–26, 2015.
- [60] C.-C. Tsai, D. E. Porter, and M. Vij, “Graphene-sgx: A practical library {OS} for unmodified applications on {SGX},” in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 645–658.

- [61] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumar, D. O’keeffe, M. L. Stillwell *et al.*, “{SCONE}: Secure linux containers with intel {SGX},” in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 689–703.
- [62] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, “Ryoan: A distributed sandbox for untrusted computation on secret data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–32, 2018.
- [63] J. Lind, C. Priebe, D. Muthukumar, D. O’Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eysers, R. Kapitza *et al.*, “Glamdring: Automatic application partitioning for intel {SGX},” in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 285–298.
- [64] D. Goltzsche, C. Wulf, D. Muthukumar, K. Rieck, P. Pietzuch, and R. Kapitza, “Trustjs: Trusted client-side execution of javascript,” in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.
- [65] H. Wang, E. Bauman, V. Karande, Z. Lin, Y. Cheng, and Y. Zhang, “Running language interpreters inside sgx: A lightweight, legacy-compatible script code hardening approach,” in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 114–121.
- [66] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, “Securestreams: A reactive middleware framework for secure data stream processing,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, 2017, pp. 124–133.
- [67] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, “Sgxbounds: Memory safety for shielded execution,” in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 205–221.
- [68] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, “Sgx-shield: Enabling address space layout randomization for sgx programs.” in *NDSS*, 2017.
- [69] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, “Eleos: Exitless os services for sgx enclaves,” in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 238–253.
- [70] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, “T-sgx: Eradicating controlled-channel attacks against enclave programs.” in *NDSS*, 2017.