UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCES AND ENGINEERING

Full Stack Protection Leveraging User-level Enclaves

by

Dimitris Deyannis

PhD Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

Heraklion, July 2023

UNIVERSITY OF CRETE

DEPARTMENT OF COMPUTER SCIENCE

Full Stack Protection Leveraging User-level Enclaves

PhD Dissertation Presented by

Dimitris Deyannis

in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

APPROVED BY:

Author: Dimitris Deyannis

Supervisor: Sotriris Ioannidis, Associate Professor, Technical University of Crete

---

Committee Member: Evangelos P. Markatos, Professor, University of Crete

---

Committee Member: Michalis Polychronakis, Associate Professor, Stony Brook University

---

Committee Member: Maria Papadopouli, Professor, University of Crete

---

Committee Member: Xenofontas Dimitropoulos, Associate Professor, University of Crete

---

Committee Member: Kostas Magoutis, Associate Professor, University of Crete

---

Committee Member: Nikos Vasilakis, Assistant Professor, Brown University

---

Department Chairman: Antonis Argyros, Professor, University of Crete

Heraklion, July 2023

”Be good to the people on your way up the ladder
’cause you’ll need them on your way down”

— Lucky Dube

Acknowledgments

I would like to thank my supervisor, Associate Professor Sotiris Ioannidis for giving me the opportunity to work on so many different, challenging and interesting projects, over the past ten years. Our collaboration has greatly contributed to my academic and technical growth. Also, the remaining members of my committee, Evangelos P. Markatos, Michalis Polychronakis, Maria Papadopouli, Xenofontas Dimitropoulos, Kostas Magoutis and Nikos Vasilakis for the valuable comments and useful feedback during my defense.

Moreover, I am thankful to Assistant Professor Giorgos Vasiliadis for his guidance during this academic journey and to Lazaros Koromilas for everything I learned from him throughout our collaboration.

I am indebted to Assistant Professor Nikos Vasilakis for his invaluable mentoring, support, advice and hospitality. His feedback and guidance have greatly contributed to this research work and to my academic growth. Also, to Dimitris Karnikis for his dedication and technical contribution to this work as well as for the best collaboration I had over the past ten years.

I also want to express my deepest gratitude to Professor Youki Kadobayashi, Associate Professor Doudou Fall and to all the members of the Laboratory for Cyber Resilience at NAIST. I will always cherish my days in Japan as the most interesting and happiest time of this academic journey.

My warmest regards to Evangelos Ladakis, Elias P. Papadopoulos, Eva Papadogiannaki, Giorgos Christou, Giorgos Tsirantonakis, Panagiotis Papadopoulos, Michalis Diamantaris, Alexander Shevtsov, Rafail Tsirbas, Nikolaos Chalkiadakis and all the other present and past members of the Distributed Computing Systems and Cybersecurity (DiSCS) Laboratory at ICS-FORTH, for their friendship, advice and commitment, as well as Christos Papachistos for his technical support.

Finally, I want to thank my family and friends, as well as Eirini Bardani, for their invaluable support and caring.

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

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

The research work was supported by the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under the HFRI PhD Fellowshipgrant (GA. No. 2767).

Abstract

Over the past few decades, computing power, storage capabilities, and network technologies have experienced exponential growth, driven by breakthroughs in hardware and software development. These advancements have brought profound transformations in various aspects of human life, making computing devices, from desktops and laptops to smartphones and embedded systems, affordable and readily available to everyday users. Also, the widespread adoption of the Internet, the emergence of mobile networks, and the advent of cloud computing have interconnected these devices like never before, revolutionizing the way information is shared, collected, stored, and analyzed.

However, these rapid technological advancements have led to the development of complex hardware and software ecosystems and come with serious challenges and concerns regarding their security and the privacy of their users. The traditional systems tasked to safeguard the various layers of modern computing systems are becoming more complex, while adversaries utilize years of experience and advanced techniques to exploit the ever-increasing attack surfaces. To address this issue, the research community and the industry propose novel systems, targeting specific use cases, that manage to achieve their goal but often are too specialized to be interoperable or adopted by the end users.

In this work, we explore the design and implementation of a modular framework that aims to raise the security bar at four core layers, providing interoperable components that can be utilized by a wide range of devices, ranging from end-user systems to cloud infrastructure. We base our work on Trusted Execution Environments (TEEs), a technology available to complex systems as well as common computing devices and prove that this platform can serve as a common base for building efficient and interoperable security systems, able to safeguard hosts, computations, communications, data management, and the security mechanisms themselves. In addition, we propose techniques to extend the capabilities of modern TEEs, enabling secure execution of unmodified applications developed with high-level languages, secure distributed execution of such applications, and leverage the sandboxing properties of TEEs to enable privacy-preserving computations in the cloud. Finally, we evaluate our work using off-the-shelf hardware with real applications and datasets to highlight the efficiency and practicality of the proposed architecture.

Supervisor: Sotiris Ioannidis
Associate Professor
School of Electrical and Computer Engineering
Technical University of Crete

Περίληψη

Τις τελευταίες δεκαετίες, η υπολογιστική ισχύς, οι δυνατότητες αποθήκευσης και οι τεχνολογίες δικτύου έχουν δεχθεί εκθετική ανάπτυξη, οδηγούμενες από τις επαναστατικές εξελίξεις στην δημιουργία υλικού και λογισμικού. Αυτή η πρόοδος έχει επιφέρει σημαντικές αλλαγές σε διάφορες πτυχές της ανθρώπινης ζωής, καθιστώντας τις συσκευές, από σταθερούς και φορητούς υπολογιστές έως τα έξυπνα κινητά τηλέφωνα και τα ενσωματωμένα συστήματα, προσιτές και εύκολα διαθέσιμες στους καθημερινούς χρήστες. Επίσης, η ευρεία υιοθέτηση του Διαδικτύου, η εμφάνιση των ασύρματων δικτύων και το υπολογιστικό νέφος έχουν συνδέσει αυτές τις συσκευές όπως ποτέ άλλοτε, επαναπροσδιορίζοντας τον τρόπο με τον οποίο κοινοποιούνται, συλλέγονται, αποθηκεύονται και αναλύονται τα δεδομένα.

Ωστόσο, αυτές οι ταχείες τεχνολογικές εξελίξεις έχουν οδηγήσει στην ανάπτυξη πολύπλοκων οικοσυστημάτων υλικού και λογισμικού και συνοδεύονται από σοβαρές προκλήσεις και ανησυχίες όσον αφορά την ασφάλειά τους και την ιδιωτικότητα των χρηστών τους. Τα παραδοσιακά συστήματα που χρησιμοποιούνται για την προστασία των διαφόρων επιπέδων των σύγχρονων υπολογιστικών συστημάτων γίνονται όλο και πιο πολύπλοκα, ενώ οι επιτιθέμενοι εκμεταλλεύονται την πολυετή εμπειρία τους και προηγμένες τεχνικές για να εκμεταλλευτούν τις αυξανόμενες επιφάνειες επίθεσης. Για να αντιμετωπίσουν αυτήν την απειλή, η ερευνητική κοινότητα και η βιομηχανία προτείνουν καινοτόμα συστήματα, που ειδικεύονται σε συγκεκριμένες περιπτώσεις χρήσης, τα οποία καταφέρνουν να επιτύχουν τον στόχο τους, αλλά συχνά είναι πολύ εξειδικευμένα για να είναι διαλειτουργικά ή για να υιοθετηθούν από τους τελικούς χρήστες.

Σε αυτήν την εργασία, εξετάζουμε τον σχεδιασμό και την υλοποίηση ενός ευέλικτου πλαισίου προστασίας που στοχεύει στην αύξηση του βαθμού ασφάλειας σε τέσσερα βασικά επίπεδα, παρέχοντας διαλειτουργικά συστήματα που μπορούν να εγκατασταθούν και να χρησιμοποιηθούν από μια ευρεία γκάμα συσκευών, από απλά συστήματα χρηστών έως υποδομές υπολογιστικού νέφους. Βασίζουμε την εργασία μας σε Ασφαλή Περιβάλλοντα Εκτέλεσης (ΑΠΕ), μια τεχνολογία η οποία είναι διαθέσιμη για πολύπλοκα συστήματα καθώς και για κοινές συσκευές. Επιπλέον, αποδεικνύουμε ότι αυτή η τεχνολογία μπορεί να λειτουργήσει ως κοινή βάση για την κατασκευή αποτελεσματικών και διαλειτουργικών συστημάτων ασφαλείας, τα οπο-

ία μπορούν να προστατεύουν τις συσκευές, τους υπολογισμούς, τις επικοινωνίες, τη διαχείριση δεδομένων καθώς και τους ίδιους τους μηχανισμούς ασφαλείας. Επίσης, προτείνουμε τεχνικές για την επέκταση των δυνατοτήτων των σύγχρονων ΑΠΕ, επιτρέποντας την ασφαλή εκτέλεση μη τροποποιημένων εφαρμογών που έχουν αναπτυχθεί σε γλώσσες υψηλού επιπέδου, την ασφαλή κατανεμημένη εκτέλεση τέτοιων εφαρμογών και την αξιοποίηση των ιδιοτήτων απομόνωσης των ΑΠΕ για την εκτέλεση υπολογισμών που διαφυλάσσουν την ιδιωτικότητα στο υπολογιστικό νέφος. Τέλος, αξιολογούμε την εργασία μας χρησιμοποιώντας κοινό υπολογιστικό υλικό με πραγματικές εφαρμογές και δεδομένα για να επιδείξουμε την αποδοτικότητα και την πρακτικότητα της προτεινόμενης αρχιτεκτονικής.

Επόπτης: Σωτήρης Ιωαννίδης
Αναπληρωτής Καθηγητής
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Πολυτεχνείο Κρήτης

Contents

Acknowledgments . . . vii
Abstract . . . ix
Abstract in Greek . . . xi
Table of Contents . . . xiii
List of Figures . . . xix
List of Tables . . . xxiii
1 Introduction . . . 1
1.1 Motivation . . . 2
1.2 Thesis Statement . . . 4
1.3 Contributions . . . 5
1.4 Outline of this Dissertation . . . 7
2 Background . . . 9
2.1 Hardware-assisted TEE . . . 11
2.1.1 Intel Software Guard eXtensions (SGX) . . . 12
2.1.2 ARM TrustZone . . . 16
2.1.3 AMD SEV-SNP . . . 18
2.1.4 Sanctum . . . 18
2.1.5 AEGIS . . . 19
2.1.6 Apple Secure Enclave . . . 19
2.1.7 Bastion . . . 20
2.1.8 Trusted Platform Module (TPM) . . . 20
2.1.9 AMD PSP . . . 20
2.1.10 Intel Management Engine (ME) . . . 21
2.1.11 x86 System Management Mode (SMM) . . . 21
2.2 Further Research on TEEs . . . 22
2.2.1 GPU-assisted TEEs . . . 23
2.2.2 FPGA-assisted TEEs . . . 24
3 Security Framework Overview . . . 25
3.1 Host Protection . . . 26
3.2 Operating System Service Protection . . . 26
3.3 Secure Communication . . . 27
3.4 Secure Dynamic Execution . . . 27
3.5 Threat Model and Assumptions . . . 28

4 Privacy-preserving Malware Analysis . . . 29
4.1 Signature-based Malware Detection . . . 29
4.2 Threat Model . . . 30
4.3 Design . . . 30
4.3.1 TrustAV Client . . . 31
4.3.2 TrustAV Server . . . 32
4.3.3 Service Registration . . . 34
4.3.4 Remote Attestation . . . 34
4.4 Implementation . . . 35
4.4.1 Malware Scanning Inside SGX Enclaves . . . 35
4.4.2 SGX Enclave I/O . . . 36
4.4.3 Performance Optimizations . . . 37
4.4.4 TrustAV Clients . . . 39
4.5 Evaluation . . . 39
4.5.1 Evaluation Setup . . . 40
4.5.2 Malware Analysis . . . 40
4.5.3 Performance Optimization Analysis . . . 42
4.6 Summary . . . 44
4.6.1 Discussion . . . 45
5 Kernel Integrity Monitoring . . . 47
5.1 Design . . . 47
5.2 Implementation . . . 48
5.2.1 Mapping Kernel Memory to SGX Enclaves . . . 48
5.2.2 Kernel Integrity Monitoring in SGX . . . 50
5.3 Evaluation . . . 51
5.3.1 Evaluation Setup . . . 51
5.3.2 Snapshot Frequency . . . 51
5.3.3 Monitoring Accuracy . . . 52
5.4 Summary . . . 54
6 Enhancing a Modern Operating System . . . 57
6.1 The Android OS . . . 58
6.2 Threat Model . . . 59
6.3 Design . . . 59
6.3.1 Trusted Execution and Storage . . . 60
6.3.2 Andromeda Services . . . 61
6.4 Implementation . . . 63
6.4.1 Setting up SGX for Android . . . 63
6.4.2 Running an SGX Application . . . 65
6.5 Andromeda Framework . . . 66

6.5.1 Andromeda Keystore . . . 66
6.5.2 Native Development . . . 67
6.5.3 Andromeda Java API . . . 67
6.6 Evaluation . . . 69
6.6.1 Security Analysis . . . 69
6.6.2 Performance Analysis . . . 70
6.7 Summary . . . 76
7 Secure and Attested Communications . . . 77
7.1 Intel SGX Remote Attestation . . . 77
7.2 Threat Model . . . 78
7.3 Design . . . 79
7.3.1 Involved Parties . . . 79
7.3.2 Protocol Design . . . 80
7.3.3 Attestation Caching . . . 81
7.3.4 Protocol Centralization . . . 83
7.4 Evaluation . . . 84
7.4.1 Evaluation Setup . . . 84
7.4.2 Performance Evaluation . . . 85
7.5 Summary . . . 86
8 Trusted Execution of High-level Dynamic Languages . . . 87
8.1 TEE Computing Challenges . . . 87
8.2 Addressing the Challenges . . . 88
8.3 Threat Model . . . 89
8.4 Architecture . . . 90
8.4.1 Interpreter Enclaves . . . 90
8.4.2 LuaGuardia Client Stub . . . 91
8.4.3 Local Execution . . . 92
8.5 Implementation . . . 92
8.5.1 Porting the LuaVM . . . 92
8.5.2 System Call Handling . . . 94
8.5.3 External Modules . . . 95
8.5.4 Maintaining Global State . . . 96
8.5.5 Code and Client Isolation . . . 97
8.5.6 Optimizations . . . 97
8.6 Evaluation . . . 98
8.6.1 Security Analysis . . . 98
8.6.2 Microbenchmarks . . . 99
8.6.3 Benchmark Applications . . . 101
8.6.4 Performance Optimizations . . . 103

8.6.5 Real-world Application 1: wrk2 . . . 105
8.6.6 Real-world Application 2: Snabb/pflua . . . 107
8.6.7 Real-world Application 3: Snabb/VPN . . . 109
8.7 Summary . . . 110
9 Automated Scaling of Trust-Oblivious Systems to TEEs . . . 113
9.1 JavaScript . . . 114
9.2 Design . . . 114
9.2.1 Atlas Components . . . 114
9.2.2 Atlas Execution Flow . . . 114
9.2.3 Atlas Workers . . . 116
9.2.4 Scaling Out . . . 116
9.3 Implementation . . . 116
9.3.1 Porting QuickJS . . . 117
9.3.2 Bootstrapping . . . 117
9.3.3 Distribution Steps . . . 118
9.3.4 External Modules . . . 119
9.3.5 System Call Handling . . . 120
9.4 Evaluation . . . 122
9.4.1 Evaluation Setup . . . 122
9.4.2 Applications . . . 122
9.4.3 Dynamic Response . . . 124
9.4.4 Scalability Characteristics . . . 128
9.4.5 Benchmarks . . . 130
9.4.6 Microbenchmarks . . . 131
9.5 Summary . . . 133
9.5.1 Discussion . . . 133
10 Related Work . . . 135
10.1 Malware Detection . . . 135
10.2 Kernel Integrity Monitoring . . . 136
10.3 Trusted Execution for Android . . . 137
10.4 Secure and Attested End-to-End Communication . . . 138
10.5 Trusted Dynamic Code Execution . . . 139
10.6 Remote Confidential Computations . . . 141
10.7 Improving Intel SGX . . . 142
11 Conclusion . . . 143
11.1 Synopsis of Contributions . . . 144
11.2 Directions for Future Work . . . 145
Bibliography . . . 147
Appendices

A Publications 167
B Acronyms 169

List of Figures

2.1 Application attack surface without and with using Intel SGX. 13
2.2 Intel SGX application execution flow. 14
2.3 Intel SGX remote attestation process. 15

3.1 Enclave-based security framework overview. 25

4.1 Architecture overview of the malware detection service. 31
4.2 An overview of the Aho-Corasick DFA serialized as an integer array. The DFA contains the patterns *{he, she, his, hers}*. In sub-figure (a), dark blue nodes indicate final states (i.e., end of a pattern) while the same information is indicated with negative values in sub-figure (b). 36
4.3 Throughput evaluation of our malware scanning engine when executed with and without Intel SGX enclaves. Six different signature sets are tested against custom input streams infected by 0%, 10%, 50% and 100%. The Y2 axis indicates the DFA size. 41
4.4 Performance evaluation of TrustAV when executed within enclaves while the automata are stored in untrusted memory in plain-text format. 42
4.5 Performance comparison of the four ruleset protection schemes when the automaton size exceeds the ECP's capacity and the infection rate is 100%. . 43

5.1 Architecture overview of the kernel integrity monitor. 48
5.2 Mapping OS kernel memory to the address space of the integrity monitor. In *step 1*, we locate a desired kernel virtual address pointing to a physical address. In *step 2*, we duplicate this mapping to user space using our page table manipulation kernel module. In *step 3*, we pass the user space virtual address to the SGX-enclave using the `user_check` option. 49
5.3 Sustainable scanning frequency using CRC-32 and SHA-256 in respect to the number of monitored regions. 52
5.4 Detection rate of the self-hiding LKM with different snapshot frequencies. Frequency configurations of 8 KHz or more achieve a 100% detection rate. For each frequency configuration we monitor the head of the kernel modules list and load a module that deletes its entry from the list 100 times. . . . 53

5.5 Scanning frequency and detection rate of the self-hiding LKM using CRC-32 and SHA-256 in respect to the number of monitored regions. 54

6.1 Andromeda architecture overview. 60
6.2 The Andromeda Keystore architecture. Cipher keys are stored only in SGX enclaves. Encryption and decryption is performed using the default Keystore API, internally redirecting to Andromeda's trusted implementation. . . 62
6.3 Throughput comparison between the AES-128 CTR found in Android's Keystore and the SGX-enabled version provided by Andromeda's Keystore. Figure (a) presents the throughput achieved by Keystore's low-level functions while Figure (b) presents the throughput perceived at the APK level. 71
6.4 Sustained throughput achieved for the vanilla and the SGX-enabled implementation of the RSA-1024 cryptographic algorithm. 72
6.5 Sustained throughput of the vanilla and SGX-enabled data transfers between the host and the securely paired device using Wi-Fi. 74
6.6 Sustained throughput of the vanilla and SGX-enabled data transfers between the host and the securely paired device using Bluetooth. 74
6.7 Performance comparison of the different Andromeda-enabled execution methods, including offloading, against the vanilla Java-based versions. 75

7.1 Assumed TCB and possibly malicious components. 78
7.2 Attestation protocol. 81
7.3 Attestation protocol with response caching. 83
7.4 Execution time frequencies. 85
7.5 Execution time comparison. 85

8.1 LuaGuardia architecture overview and secure execution life cycle. 90
8.2 Performance comparison between the vanilla LuaVM and our SGX-protected LuaVM when reading a 32MB file with variable read buffer sizes. 100
8.3 Performance comparison between the vanilla LuaVM and the SGX-enabled runtime when performing one million random accesses (read/write) to memory locations ranging between 1KB and 4MB. 101
8.4 Execution time achieved by the vanilla LuaVM and LuaGuardia when executing 12 cryptographic benchmarks. The overhead introduced by LuaGuardia is indicated on top of each bar. 102
8.5 End-to-end performance comparison of the vanilla LuaVM with LuaGuardia's two operation modes when executing 12 Lua benchmarks. 103
8.6 Execution time breakdown with and without initialization optimizations. . . 104
8.7 Performance comparison between the vanilla LuaVM and LuaGuardia with and without system call batching. 105

8.8 Performance comparison between the vanilla LuaVM and LuaGuardia with and without optimizations when executing 12 Lua benchmarks. 106
8.9 Performance comparison between the vanilla LuaVM and LuaGuardia using wrk2 with three benchmarks. 107
8.10 Performance comparison between the vanilla LuaVM and LuaGuardia using Snabb/pflua with 100 user-defined rules. 108
8.11 Performance comparison between the vanilla LuaVM and LuaGuardia using our custom Snabb/VPN module. 110

9.1 Atlas architecture and execution life cycle overview. 115
9.2 Dynamic response capabilities when executing the contact discovery application on the multiprocessor and the cloud with variable load spikes. 126
9.3 Dynamic response capabilities during the first event (E0) when executing the contact discovery application on both setups. 127
9.4 Dynamic response capabilities when executing the real-world applications using Azure cloud instances with variable load spikes. 128
9.5 Atlas speedup gain on Azure cloud using up to eight worker instances when executing real-world applications. 129
9.6 Performance comparison between the vanilla QuickJS implementation and Atlas with a single worker node when executing various benchmarks. 130
9.7 Performance comparison between the vanilla QuickJS and Atlas when reading a 20MB file with variable read buffer sizes. 131
9.8 Performance comparison between the vanilla QuickJS and our SGX-enabled runtime when performing one million random accesses (read/write) to memory locations ranging between 64B and 2MB. 132

List of Tables

2.1 Trusted execution environments. 12
2.2 Some of the available TrustZone TEEs (secureOS). 17

6.1 Andromeda Java API for SGX enclave utilization. 68

7.1 Execution time statistics in milliseconds. 85

8.1 Functions issuing system calls, ported to the enclave-protected Lua runtime. 95
8.2 Benchmark algorithms and their operation. 103

Chapter 1
Introduction

With the rapid technological advancement in the areas of the Internet of Things (IoT), mobile and smart devices, and the growing reliance on digital connectivity in various industries, the number of devices and connections raises exponentially. This trend is expected to persist as technology continues to advance and more devices become interconnected, resulting in a greater need for robust security measures to protect devices and connections from potential cyber threats. More specifically, Cisco's forecast [62] predicts that the average number of devices per capita will grow from 2.4 in 2018 to 3.6 by 2023, while the share of Machine-To-Machine (M2M) connections (also referred to as IoT) will grow from 33% to 50%. In addition, the percentage of usage of mobile devices grows and it is expected that over 70% of the global population will have mobile connectivity by 2023. In the meantime, cyber threats are constantly evolving, bypassing state-of-the-art countermeasures. For instance, Symantec's Threat Hunter Team reported [182] that "*Ransomware groups these days employ quite a diverse toolset, making use of a mixture custom malware, legitimate software, and operating system features (also known as living off the land)*".

The protection of the higher levels of a system's software stack is traditionally achieved using malware analysis tools, Network Intrusion Detection Systems (NIDS), permission control mechanisms, data tracking, etc. However, such systems are nowadays very complex and require powerful processors to handle modern attacks, yielding high power consumption. Recently, the advent of cloud computing has led to the outsourcing of many middlebox applications, including deep packet inspection and virus scanning. The advantages are numerous, such as the ability to be utilized by almost every end-user device, lower costs spent on equipment, operation, and maintenance and better performance and scalability. However, offloading functionality and sensitive personal data (e.g., user network traffic or files) to a possibly untrusted third-party entity automatically broadens the attack surface and aims to solve the security and performance problems without taking user privacy into account.

One of the most crucial security problems to completely safeguard devices and applications is to ensure the integrity of the Operating System (OS), as it has a direct impact

to the security of the executing processes and their data. At the same time, adversaries and attackers are becoming more and more powerful, utilizing years of experience and advanced tools, creating attacks that stay under the radar of modern anti-malware systems. While various works aim to protect operating systems against sophisticated rootkits, they often utilize external hardware or hypervisors. Such solutions cannot always be deployed on end-user devices since the necessary hardware is usually complex and non-commodity or the required hypervisor is not available.

These limitations also apply for the Android OS which has become a very popular open-source operating system, targeting a large variety of devices. Mobile devices play a core part of our everyday life and usually process and store a vast amount of privacy-sensitive data, such as personal information, financial accounts, cryptographic keys and high-profile enterprise assets. Android is also used as a hub for a diverse set of smaller devices, such as wearables and IoT, sending their data to a corresponding application that runs on an Android device. As such, protecting the operating system, the applications, and enabling them to compute on sensitive data that external devices generate, while preserving their integrity and preventing any unwanted or malicious modifications of these data, is an important problem.

However, regardless of the available systems guarding the host and its communication channels, the need to secure the executing applications still remains. One of the most popular approaches to this issue is enabling confidential computing with 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-unsafe 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, recompilation, and extra linking steps.

1.1 Motivation

While various works exist in the areas of kernel integrity monitoring, malware detection and secure code execution, they oftentimes introduce heavy system modifications or external/custom hardware. These modifications are usually not trivial or assume the existence of hardware devices or architectures that cannot be found in commodity systems. Also, these properties render them non-interoperable, preventing the ability to combine multiple solutions on a single host to protect various software levels.

In the area of mobile devices, developing security tools for Android, especially based on TrustZone [39], is even more complicated despite the fact that it is available as a feature for almost all ARM CPUs. Many times it cannot be directly used since it requires control

of the device's firmware, which is not always available, preventing TrustZone-based software to be seamlessly deployed across different devices. This leads to the development of custom solutions that target a single device, usually a prototype or a development kit. For these reasons, while most proposed solutions manage to mitigate the security issues they are addressing and provide useful research results, they rarely get commercially adopted.

Leveraging trusted execution environments appears to be a very appealing approach for building a protection stack based on a common technology with interoperable subcomponents. This stack can address various security issues across the entire system's software layers and can be executed on off-the-shelf systems without modifications. Also, this approach allows the extension of such a stack by building upon established, tested, and documented libraries and APIs.

The currently available trusted execution environments offer a plethora of mechanisms for handling some very important security issues. First, deploying security mechanisms inside enclave-like TEEs offers protection for the mechanisms themselves; an aspect that is not comprehensively discussed in the literature. This is a realistic threat as many real-world attacks are known to disable or disrupt the execution of the protection mechanisms to achieve their goal — an isolation property that is usually achieved with custom hardware, hardware modifications or hypervisors. Also, secure enclaves are able to store or seal sensitive and private data and provide mechanisms for integrity checking upon reuse. Another benefit of building a security stack based on enclaves is the ability to utilize local and/or remote attestation. In this way, the various components of the security system can be attested via a local service or a remote trusted party. Finally, many capabilities of TEEs are not yet thoroughly explored by the literature, such as (i) repurposing their functionality to address the privacy issues raised by security software, especially when executed remotely, (ii) the option of enabling dynamic or type-safe code execution, as well as (iii) their potential to assist in establishing CPU-to-CPU communication channels with stronger security guarantees than mechanisms performing encryption using unprotected memory spaces.

For this work, we choose to utilize Intel's x86 platform as it provides, up to this date, the most versatile and extendable user-level enclave-based trusted execution environment, namely Intel Software Guard eXtensions (SGX) [67]. The work proposed in this thesis is not directly dependent on SGX or the x86 architecture and the proposed paradigms can be implemented on any TEE platform that offers user-level enclaves and attestation capabilities, such as SANCTUARY [55], which offers enclaves based on TrustZone. However, at this point, neither SANCTUARY nor any other enclave-based TEE (other that Intel SGX) is available to developers and since SANCTUARY is based on TrustZone, it requires access to the device's firmware to be deployed, thus it cannot be seamlessly installed across devices.

SGX enables the creation of secure and hardware-assisted isolated software containers in the user space, called enclaves, whose code and data cannot be read or modified by

any other process aside from the one utilizing them. This reverse-sandbox property also restricts the OS kernel or debuggers from reading or tampering their contents. Intel SGX utilizes hardware assisted encryption, performed by the CPU, using a supplied on-chip mechanism called Memory Encryption Engine (MEE). The MEE is responsible for encrypting a portion of the live memory and providing it to SGX where the data and the code of several enclaves may reside. Data from the trusted enclave are decrypted on the fly during execution within the CPU and are accessible only by the enclave. When enclave memory pages need to be swapped out and moved to the DRAM, SGX encrypts them using the MEE. As a result, every process is restricted from accessing the enclave's contents. Also, enclave data can be securely sealed and exported in the untrusted file system in an encrypted format, accompanied by metadata used for integrity checking upon reuse. With these mechanisms, SGX protects the confidentiality of the enclave pages, even when assuming an untrusted or even a malicious operating system, hypervisor or firmware. These properties render SGX the best of the available TEE platforms to build this work on.

1.2 Thesis Statement

In this work we **prove that user-level enclave-based Trusted Execution Environments can be leveraged to implement a modular security framework consisting of interoperable components, able to protect various software layers and provide user privacy while operating with off-the-shelf software and hardware infrastructure**. To address the multiple security issues discussed above, we develop such a framework and partition it into four different layers: (i) *host protection*, (ii) *operating system service protection*, (iii) *secure communications*, and (iv) *secure dynamic code execution*. In total, our stack provides six components: (i) a *malware detection engine*, (ii) a *kernel integrity monitor*, (iii) a *service for providing secure and attested end-to-end communications*, (iv) a *TEE-enhanced Android OS*, (v) two *protected high-level language runtimes* that enable secure dynamic code execution, and (vi) a *scheduling system* that dynamically scales high-level code execution over multiple TEEs, aiming to cover all aspects of modern system security.

Our security framework includes a signature-based malware detection solution, deployable either locally or on remote infrastructure, that operates without compromising user privacy by exposing sensitive files and data to infrastructure providers. The framework also provides a lightweight snapshot-based kernel integrity monitor able to detect persistent or transient rootkits without the need for external hardware or hypervisors.

To cover the field of mobile devices, we extend our security stack by porting the Intel SGX SDK and PSW to the Android x86 platform and implement the first, to our knowledge, SGX-enabled Android OS. We design simple APIs and a custom cross-compiler toolchain for developers that wish to extend our security stack or implement new tools based on enclaves, able to be deployed across different mobile devices without requiring firmware

flashing or other intrusive modifications.

We also extend the abilities of secure enclaves to overcome their limitations and repurpose their functionality. An idea not proposed in the literature yet, is the deployment of such reverse-sandboxes in the cloud to ensure user privacy. By establishing secure end-to-end communication channels between the end-user device and the remotely executed enclave, the users can offload sensitive data and applications without the risk of honest but curious providers taking access to them. In this way, we handle the offloading privacy risks that are often not taken into consideration by academic or commercial works.

Finally, we further extend our framework beyond providing interoperable malware detection tools, OS enhancements and protected communication. While the TEEs offered by the major vendors provide a large set of security features, they mandate secure code development in C/C++. This property does not allow for dynamic code execution or application development in a modern high-level and type-safe language. Also, the integration of a TEE technology in a new or existing software ecosystem requires heavy modifications and expertise on codebase partitioning. With this work, we address the common limitations of available TEE technologies which prevent them from being widely adopted in modern software ecosystems, built using high-level type-safe languages. The main idea for reaching this goal is to treat code as data, enabling it to be transferred to and from the enclave in an attested way and then execute it in the secure and attested environment. Building on this idea, we provide dynamic software execution inside enclaves by utilizing custom-ported popular language runtimes, such as Lua and JavaScript. The combination of secure dynamic execution and attested communication also enables the trusted execution via offloading for devices not equipped with TEE-capable hardware.

1.3 Contributions

More specifically, this work contributes the following in the areas of (i) *malware detection*, (ii) *kernel integrity monitoring*, (iii) *trusted execution for Android*, (iv) *secure and attested end-to-end communications*, (v) *secure dynamic code execution*, and (vi) *secure distributed dynamic code execution*:

Malware Detection

- A practical cloud-based malware detection solution that provides users with strong privacy-preserving guarantees regarding the processing of their personal and sensitive data remotely by utilizing hardware assisted enclaves.
- Our work mitigates the performance constraints introduced by user-level enclaves, specifically for signature-based detection approaches, such as malware and intrusion detection systems.

Kernel Integrity Monitoring

- A kernel integrity monitor that leverages user-level enclaves to protect its code and data from identification and modification.
- Proof of the effectiveness of the proposed system in identifying transient kernel-side attacks as well as a study of the appropriate monitoring intervals that guarantee that transient rootkits are not able to perform any malicious actions and remain undetected at the same time.

Trusted Execution for Android

- A systematic methodology that can be used to port the SGX framework to the Android OS, including the SGX kernel driver, the required libraries and background services needed for its operation and a custom cross-compiler. This work has lead to the first and only, to our knowledge, SGX-enabled Android OS.
- Popular Android services, enhanced with SGX capabilities and a programming paradigm tailored for externally paired devices that enables efficient and trusted data and code flow between external devices that pair with the SGX-enabled Android OS.

Secure and Attested Communications

- A system that provides seamless establishment of enclave-to-enclave (CPU-to-CPU) attested and encrypted network communication between two or more parties.
- A caching system for SGX RA responses that reduces the latency of consecutive connections, rendering them comparable or faster than a standard TLS handshake.

Secure Dynamic Code Execution

- Two lightweight secure execution systems that enable confidential computing using high-level languages, namely Lua and JavaScript, eliminating the need to learn or port code to device-specific TEEs.
- Implementation and evaluation of several low-level optimizations, such as enclave pre-allocation and reuse, as well as runtime-specific optimizations to accelerate their performance and allow transparent execution of legacy code without modifications.

Secure Distributed Dynamic Code Execution

- A scheduling middleware and a tag interpreter that enable the secure distributed execution of tagged functions across multiple server nodes hosting instances of TEE-protected runtimes.

- Proof that the proposed systems can be used to securely execute and distribute a diverse set of new and legacy applications, from simple popular benchmarks to real-world toolkits, with little-to-zero code modifications.

1.4 Outline of this Dissertation

This dissertation is structured as follows. Chapter 2 presents an overview of the currently available TEE solutions and discusses their various characteristics in detail. The information provided in this chapter also explains our choice for building the proposed framework based on Intel SGX. Chapter 3 offers a high-level overview of our framework's components and the software layer that each one of them targets, as well as our framework's general threat model. Then, Chapters 4 to 9 provide detailed descriptions regarding the design and implementation of each component composing our security framework. Also, these chapters offer thorough evaluation of the discussed components and a summary of the contributions derived by the research work that led to their development. Chapter 10 surveys previous work related to the field that each framework's component addresses as well as related work in the general field of trusted execution environments. Finally, Chapter 11 summarizes the key points of this dissertation and offers some directions for future work. A list of publications produced so far by the activities related to this dissertation is presented in Appendix A.

Chapter 2
Background

A Trusted Execution Environment (TEE) is a secure, trusted, and integrity-protected environment, offering secure execution, memory, and storage capabilities, usually assisted by dedicated hardware. However, the term TEE is redefined many times through the years, depending on the marketing purposes of each vendor and the scope it is used in. So far, there is no common and precise definition of the term and the properties of each available TEE are redefined with each implementation. To emphasise on this, we present the seven most cited definitions of the term, in chronological order, spanning two decades.

1. Garfinkel et al., *Terra*, 2003 [86]: A TEE is a dedicated closed virtual machine that is isolated from the rest of the platform. Through hardware memory protection and cryptographic protection of storage, its contents are protected from observation and tampering by unauthorized parties.

2. OMTP, *Advanced Trusted Environment*, 2009 [140]: A TEE resists against a set of defined threats and satisfies a number of requirements related to isolation properties, life cycle management, secure storage, cryptographic keys and protection of application code.

3. GlobalPlatform, *TEE System Architecture*, 2011 [150]: A TEE is an execution environment that runs alongside but isolated from the device's main operating system. It protects its assets against general software attacks. It can be implemented using multiple technologies and its level of security varies accordingly.

4. Vasudevan et al., *Trustworthy Execution on Mobile Devices*, 2014 [194]: The set of features intended to enable trusted execution are the following: (i) isolated execution, (ii) secure storage, (iii) remote attestation, (iv) secure provisioning, and (v) trusted path.

5. Sabt et al. *Trusted execution environment*, 2015 [157]: A TEE is a tamper-resistant processing environment that runs on a separation 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 resists against all software attacks as well as the physical attacks performed on the main memory of the system. Attacks performed by exploiting backdoor security flaws are not possible.

6. GlobalPlatform, *TEE System Architecture V1.3*, 2022 [151]: At the highest level, a Trusted Execution Environment (TEE) that meets the TEE Protection Profile (TEE PP) is an environment where the following are true: (i) All code executing inside the TEE has been authenticated. (ii) Unless explicitly shared with entities outside the TEE: (ii.a) The ongoing integrity of all TEE assets is assured through isolation, cryptography, or other mechanisms. (ii.b) The ongoing confidentiality of the contents of all TEE data assets is assured through isolation or other mechanisms such as cryptography. Data assets include keys. (iii) TEE capabilities, such as isolation or cryptography, can be used to provide confidentiality of the TA code asset. (iv) The TEE resists known remote and software attacks, and a set of external hardware attacks. (v) Both code and other assets are protected from unauthorized tracing and control through debug and test features.

7. Evervault, *What is a Trusted Execution Environment?*, 2023 [78]: A Trusted Execution Environment (TEE), also known as a Secure Enclave, is a highly constrained compute environment that allows for cryptographic verification (attestation) of the code being executed. TEEs are designed with no persistent storage, no shell access, and no network connectivity by default. As a result, they provide a completely isolated environment with heavily restricted external access, making it possible to run sensitive workloads securely. Attestation proves that a TEE is running the exact program that was selected and that its code has not been tampered with.

As we observe from the aforementioned definitions of a TEE, the general common points are (i) a TEE should be able to provide a secure execution environment for sensitive code, (ii) secure persistent data storage, (iii) a form of attestation and (iv) facilities to protect the system against external factors that could compromise its integrity. However, the first, third and seventh definitions do not include secure storage in the set of properties, something that we regard as essential. Also, the second definition indicates a set of defined attacks while other definitions do not. Moreover, the first definition describes the TEE as a "*closed virtual machine*" while the fifth and sixth definitions imply hardware assistance or hardware as part of the TCB.

2.1 Hardware-assisted TEE

As described above, there are multiple definitions of a TEE and in this work we consider that the fundamental properties of a TEE should include (i) a Secure Execution Environment (SEE), (ii) secure persistent storage, (iii) attestation mechanisms for the executed software and the underlying hardware platform, and (iv) mechanisms that, up to a certain degree, can verify the integrity of the overall system. While software-only TEEs exist in the literature, we do not elaborate on solutions following the "*secure VM*" or "*secure kernel*" approach as their scope is quite limited. Instead, we focus on hardware-assisted TEEs as they are able to provide a more robust set of features, cover a larger attack surface and allow the development of more sophisticated software solutions. Also, we do not elaborate on solutions built upon an existing technology such, as Scone [41], as they rely on existing hardware-assisted TEE technologies, such as Intel SGX.

A list of some of the most popular hardware-assisted TEEs, currently available, is presented in Table 2.1. **ISA** indicates the Instruction Set Architecture that each technology supports. **IE** indicates whether the target TEE supports Isolated Execution, **DS** indicates secure Data Storage and **RA** indicates support for Remote Attestation. Moreover, **SCP** indicates if the TEE provides mechanisms for Side-Channel attack Protection and **M/BP** for Memory or Bus Probing. The **AR** column indicates if the TEE is purposed for Academic Research. The Usability of each solution is represented with **W** for widespread, **S** for Seldom used and **U** for Unused. Finally, all TEEs are sorted by their protection **Ring** level with each level indicating the following.

- **Ring 3**: The TEE hardware allows for securing user space applications without the need to trust a privileged operating system running at Ring 0 or below.
- **Ring 0**: The TEE hardware aims to implement secure execution environments at the operating system level.
- **Ring -1**: The TEE hardware provides mechanisms to instantiate a secure stack based on a trusted hypervisor.
- **Ring -2**: The TEE hardware is implemented in the processor and is also able to operate below the hypervisor.
- **Ring -3**: The TEE hardware relies on independent coprocessors.

In this section, we particularly focus on Intel's SGX and AMD's TrustZone as they provide the most of the security properties that we think are essential for a TEE, they cover the two most popular ISA's, and their implementation can be found on the vast majority of commercially available CPUs and devices. They also provide, to an extend, SDKs for application development targeting popular operating systems, such as Linux distributions

Table 2.1: Trusted execution environments.

TEE	Ring	ISA	IE	DS	RA	SCP	M/BP	U	AR
Intel SGX	3	x86_64	Y	Y	Y	N	Y	W	N
Sanctum	3	RISC-V	Y	Y	Y	Y	N	U	Y
AEGIS	0	n/a	Y	Y	Y	N	Y	U	Y
AMD SEV-SNP	-1	x86_64	Y	N	N	N	Y	S	N
Bastion	-1	UltraSPARC	Y	Y	N	N	Y	U	Y
AMD PSP	-2	x86_64	Y	Y	N	N	n/a	W	N
ARM TrustZone	-2	Arm	Y	N	N	N	N	W	N
SMM	-2	x86	Y	N	N	N	N	S	N
Apple Secure Enclave	-3	ARM	Y	Y	N	n/a	Y	W	N
TPM	-3	n/a	N	Y	Y	n/a	n/a	W	N
Intel ME	-3	x86_64	Y	N	N	n/a	n/a	S	N

and Windows, and there is a lot of published and ongoing academic research, as well as commercial interest for their applications.

The rest of this chapter (Section 2.1.1 through Section 2.1.11) elaborates on each TEE presented in Table 2.1. Section 2.2 discuses other available hardware-assisted TEEs that either target embedded devices, are academic works with unpublished codebases or are industry proposals with closed codebases. Moreover, some of them require heavily modified hardware or software stacks or target ISAs with limited popularity. Finally, Sections 2.2.1 and 2.2.2 present TEE solutions based on Graphics Processing Units and FPGAs.

2.1.1 Intel Software Guard eXtensions (SGX)

Intel Software Guard Extensions (SGX) [67] is a set of security instructions offered by modern Intel x86 CPUs, firstly introduced with the Skylake family of processors. These instructions provide secure and hardware-assisted isolated software containers, called *enclaves*, whose code and data cannot be read or modified by any other process aside from the one utilizing them. This reverse-sandbox property also restricts the OS kernel or debuggers from reading or tampering their contents. An example of the reduced attack surface provided by SGX enclaves is depicted in Figure 2.1. Intel SGX utilizes hardware-assisted encryption, performed by the CPU, using a supplied on-chip mechanism called Memory Encryption Engine (MEE). The MEE is responsible for encrypting a portion of the live memory and providing it to SGX where the data and the code of several enclaves may reside. Data from the trusted enclave are decrypted on the fly during the execution within the CPU and are accessible only by the enclave. When enclave memory pages need to be swapped out and moved to the DRAM, SGX encrypts them using the MEE. As a result, every process is restricted from accessing the enclave contents. The available live memory

of Intel SGX enclaves ranges between 64MB and 128MB and is defined by BIOS settings. However, this does not limit the developer from accessing more memory by utilizing swapping. Yet, memory page swapping is only available for Linux while Intel's SGX driver does not offer this functionality on Windows, limiting the available usable memory to 128MB. Also, enclave data can be securely sealed and exported in the untrusted file system in an encrypted format, accompanied by metadata used for integrity checking upon reuse.

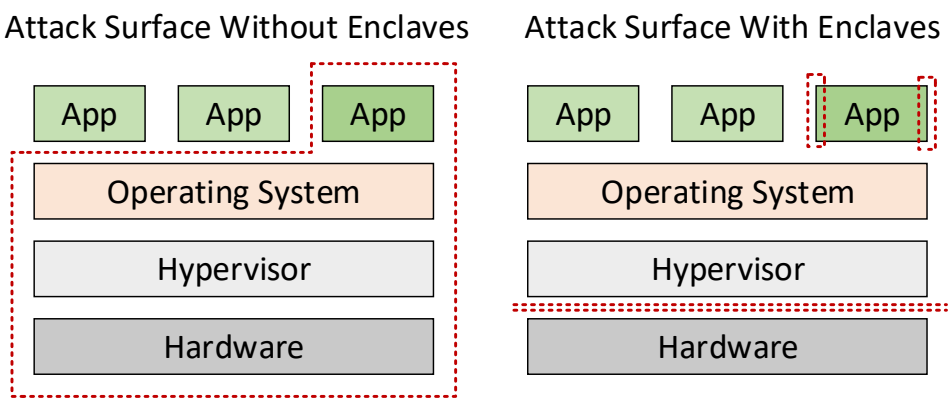


Figure 2.1: Application attack surface without and with using Intel SGX.

SGX Application Life Cycle

A typical SGX application consists of two parts: (i) the untrusted application that resides in the untrusted OS and communicates with the enclave and (ii) the secure enclave that may be bound to one or multiple applications. Communication between the two is achieved by specific functions and APIs that are declared in SGX Enclave Definition Language (EDL) during the software development and cannot be modified or extended after compilation and enclave signing. Enclaves are prohibited from directly performing undeclared I/O, accessing any system calls, or invoke privileged instructions, since the host OS kernel cannot be trusted and is rendered inaccessible. The developer has to proxy such requests to the untrusted part of the application. Such calls, known as `OCALL`s, transfer the execution outside of the secure enclaves and can only be invoked by the enclaves. Similarly, an application can perform `ECALL`s, which transfer the execution from the untrusted application to the trusted enclave, in predefined entry points, invoking a predefined enclave function. Both `ECALL`s and `OCALL`s are defined in the EDL file during the application's development and cannot be modified afterwards. The typical execution flow of an SGX enabled application is presented in Figure 2.2.

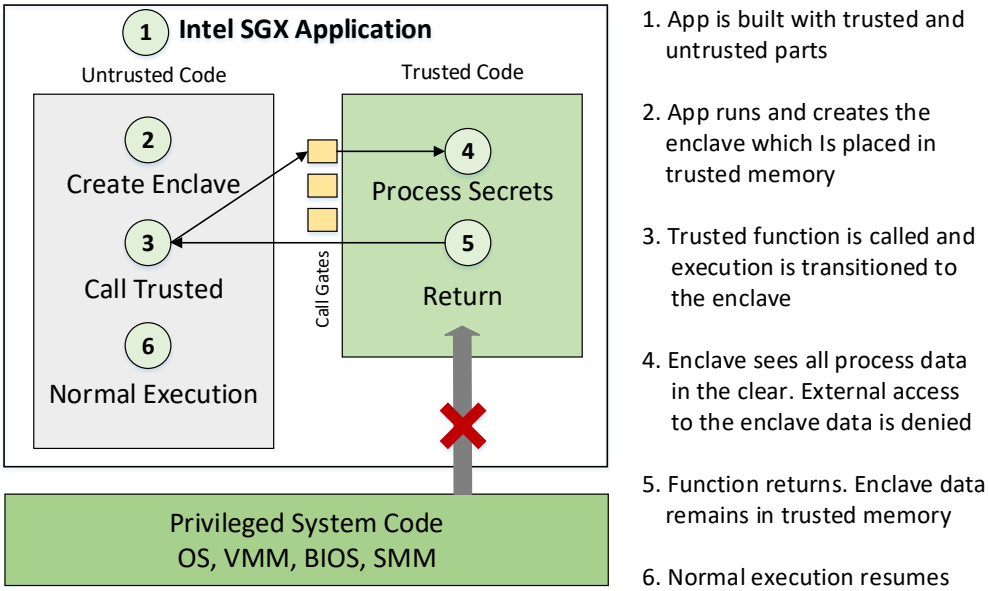


Figure 2.2: Intel SGX application execution flow.

SGX Remote Attestation

Remote attestation is the process of verifying the authenticity of a software component, running inside an isolated container, to some remote party. In the case of SGX, the software being attested is a secure enclave created by the trusted CPU hardware. For the remote attestation procedure, the CPU generates a measurement for the attested enclave which uniquely identifies it. This information is then signed by the privileged Quoting Enclave, resulting in an attestation signature (QUOTE).

Intel has access to the SGX hardware attestation key that signs the measurement. The attestation signature is generated using the EPID group signature scheme [104] to preserve privacy. The communication between the two enclaves must also be done in a secure way. This is achieved by performing a local attestation between the two communicating enclaves as a means to establish a secure channel.

The attestation signature can then be sent to the remote party, who will relay this information to the Intel Attestation Service (IAS) to verify its validity. Thus, the remote party can be aware if the enclave has been tampered or if the attested software is not running within a genuine hardware-assisted SGX enclave. This information is critical as it verifies that the SGX enclave is executed on SGX-enabled hardware and not in simulation mode, which makes the enclave accessible by debugging utilities. The SGX Remote Attestation process utilizes a modified SIGMA [113] protocol, therefore at the end of the process the remote party and the enclave establish a shared secret for secure communication. An overview of

the Intel SGX attestation process is shown in Figure 2.3.

Contrary to Trusted Platform Module (TPM), SGX Remote Attestation has the benefit that attested software runs within the CPU, thus achieving better performance. Moreover, SGX utilizes an EPID group signature scheme and attested enclaves cannot be uniquely linked back to a specific CPU through their attestation signature.

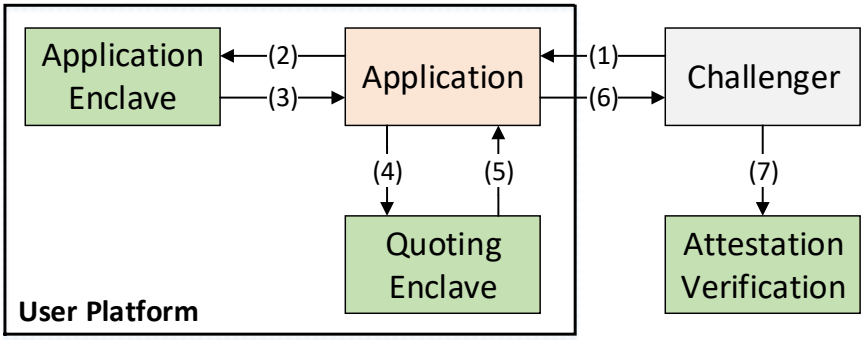


Figure 2.3: Intel SGX remote attestation process.

Intel SGX Version 2

Intel SGX2, is the latest version of Intel SGX which provides new key features that enhance the functionality of the technology compared to its predecessor. Some notable features include the provision of more live protected memory (increased to 512GB compared to 128MB for SGX1), enabling larger secure enclaves and facilitating the execution of more complex and memory-bound applications. Also, SGX2 enables permission modifications of regular enclave pages, support for dynamic addition of regular enclave pages, support for removing pages from an enclave and expanding an enclave to allow dynamic thread creation. This feature is called Enclave Dynamic Memory Management (EDMM).

Furthermore, SGX2 introduces Flexible Launch Control (FLC), a feature that allows the platform owner to control which enclaves are launched instead of Intel, substituting Launch Enclave for one not signed by Intel. This includes which enclaves are granted access to the Platform Provisioning Identifier (PPID) used with the Certificate Retrieval Service. The enclave requesting access to the PPID can be signed by the attestation Service Provider. One of the purposes of the Launch Enclave is to prevent abuse of the PPID in privacy sensitive environments and with FLC, the Launch Enclave can be written by other companies (other than Intel) but must be signed with the key corresponding to the one locked in the MSR. The MSR can also stay unlocked and then it can be modified at run-time by the VMM or the OS kernel.

With the Linux kernel version 6.2, SGX2 enclaves can use the Asynchronous Exit (AEX) Notification mechanism offered with the latest Intel CPUs. The AEX Notify path allows for running a handler on `exit` events that in turn can mitigate issues like the SGX-Step vulnerability [190]. This feature aims to toughen the defenses around Intel's SGX against an entire class of attacks.

Despite the improvements introduced with SGX2, we base our work on SGX1 for multiple reasons. First, SGX1 was the only version available to us during the development of most of the work presented in this dissertation, as SGX2 was only recently released and mainlined with Linux kernel version 6.0 [10]. Second, most of the available hardware supporting SGX enclaves is only compatible with SGX1, as SGX2 instruction extensions are only supported by some (mainly Intel Xeon Scalable) CPUs. Also, Being able to implement this work despite the various constrains of SGX1 proves that such a framework can be designed even for enclave-based TEEs with very limited secure memory capabilities and operate on commodity hosts. Utilizing SGX2 for our framework in the future will directly benefit our work as the increased memory limits and ability for dynamic thread creation can further enhance the performance and scalability of its modules.

2.1.2 ARM TrustZone

ARM TrustZone [34] is a hardware feature that enables physical separation of different execution environments and was introduced with ARMv6. TrustZone provides two environments called *secure world* or Trusted Execution Environment (TEE), and *normal world* or Rich Execution Environment (REE) respectively. Processes executed in the *secure world* are called Trusted Applications (TAs) while there are various TEE frameworks available, used to develop TAs, with OP-TEE [18] being the most popular. A list of some available TrustZone TEEs, also called *secureOS*'s is presented in Table 2.2.

The complete isolation between the two worlds is achieved by security extensions, provided by TrustZone for the various hardware components, such as the CPU die, memory, and peripherals. The CPU on a TrustZone enabled ARM platform has two security modes, the *secure mode* and the *normal mode*. Each processor mode has its own memory access region and privilege. The code running in *normal mode* cannot access the memory in the *secure mode* while the code executed in the *secure mode* can access the memory in *normal mode*. The *secure* and *normal* modes can be identified by reading the NS-bit in the Secure Configuration Register (SCR), which can only be modified in the *secure mode*.

ARM involves different Exception Levels (EL) to indicate different privileges in ARMv8 architecture and lower EL owns lower privilege. The EL3, which is the highest EL, serves as a gatekeeper, managing the transitions between the *normal mode* and the *secure mode*. The *normal mode* can trigger an EL3 exception by calling a Secure Monitor Call (SMC) instruction or by triggering secure interrupts to switch to the *secure mode*. On the other

Table 2.2: Some of the available TrustZone TEEs (secureOS).

secureOS	**Platform**	**Licence**
OP-TEE [18]	TrustZone	BSD
TLK [29]	NVIDIA Tegra	BSD
Trusty [3]	TrustZone	Apache 2.0
Open TEE [128]	TrustZone	Apache 2.0
OpenEnclaves [17]	SGX & TrustZone	MIT
Kinibi [187]	TrustZone	Commercial
TEEGRIS [159]	TrustZone	Commercial
QSEE [155]	TrustZone	Commercial

hand, the *secure mode* uses the Exception Return (ERET) instruction to switch back to the *normal mode*. Recent Cortex-A processors support SMC calls by the kernel in the *normal world*. Entry to a different world is performed on a core basis, thus limiting the parallel execution of TAs to the number of available cores. Also, it can be called from user space processes residing in the REE or from other TAs. The latter is particularly useful in order to reduce code duplication and to keep the TA's attack surface minimal. Data is passed back and forth between worlds by memory pointers or direct copies. Moreover, there are two types of hardware interrupts, (i) Interrupt Requests (IRQs) and (ii) Fast Interrupt Requests (FIRQs). Both of them can be configured as secure interrupts by configuring the IRQ and FIQ bits in SCR respectively. The secure interrupt is directly routed to the secure EL3, ignoring the configuration of the *normal world*. ARM recommends that the IRQ-bit is used as the interrupt source of the *normal world* and the FIQ-bit is used as secure interrupt.

TrustZone uses a Memory Management Unit (MMU) mechanism to support virtual memory address spaces in both the *secure* and *normal* worlds. The same virtual address space in the two worlds is mapped to different physical regions. The MMU is *secure*-world aware, and secure and non-secure descriptors are stored alongside each other. The differentiation is done by the Non-secure TLB ID (NSTID) which is an extra TLB bit. The TAs must fit in the on-chip memory. However, the secure memory is quite costly and it only ranges from 3MB to 5MB, limiting the secure code's memory even further than SGX. For this reason, TAs are expected to have small memory footprints and only contain the minimal subset of features required, aiming to reduce the TCB and its attack surface.

TrustZone also incorporates a key manager which starts with a device-specific key named Secure Storage Key (SSK). This key is derived from two pieces of information, unique to each device's processor, (i) the chip identifier and (ii) the hardware key. The TA Storage Key (TSK) is a per TA key that can be derived from the SSK and the TA's UUID. The File Encryption Key (FEK) is a per file key that can be generated upon file creation. It is used to protect the file contents, including its metadata, and is encrypted using the TSK.

Since TrustZone provides a key manager, it is also able to support secure persistent data storage for TAs (must be implemented and provided by the TA), with objects/files stored on disk in an encrypted format and signed for anti-tampering verification. TAs can access the objects in clear-text format since the TEE layer runs the cryptographic stack transparently. The securely stored files can have a unique numeric name based on a counter and an encrypted index of objects/files is maintained alongside them. Operations on the index are atomic, ensuring integrity protection by means of a hash-tree data structure that guards the index. To protect against storage replay attacks, an embedded Multi-Media Card (eMMC) storage device is required, which is a non-volatile and non-removable solid-state means of storage. This security feature is entirely implemented in the eMMC storage in the form of Replay Protected Memory Block (RPMB).

2.1.3 AMD SEV-SNP

AMD SEV-SNP (Secure Nested Paging) [35] is a security feature designed to enhance the protection of virtualized environments. SEV-SNP extends the existing AMD Secure Encrypted Virtualization (SEV) [105] technology to provide enhanced memory protection for virtual machines. SEV-SNP isolates virtual machines from each other and the hypervisor and protects them from both software and hardware attacks. The system is designed to protect virtual machines against attacks such as speculative execution attacks, side-channel attacks, and hypervisor vulnerabilities. It achieves this by encrypting virtual machine memory and isolating it from the rest of the system. It also uses hardware-enforced memory protection to prevent unauthorized access to virtual machine memory.

SEV-SNP builds on the existing SEV technology by introducing nested page tables, allowing for more efficient memory management in virtualized environments. It also provides a hardware-based root of trust for virtual machine encryption keys, which ensures that only authorized virtual machines can access their encrypted memory. Additionally, SEV-SNP provides support for live migration of encrypted virtual machines, allowing them to be moved between physical hosts while remaining encrypted and protected.

2.1.4 Sanctum

Sanctum [68] proposes a design similar to Intel's SGX, providing isolation for enclaves at user-level, using minimal hardware modifications and a trusted software component called *security monitor*. Unlike SGX, Sanctum cannot prevent DRAM attacks since there is no Memory Encryption Engine (MEE). Each Sanctum enclave manages its own page tables and page faults, and is assigned with a separate DRAM region corresponding to distinct sets in the shared Last-Level Cache (LLC) to protect against software side-channel attacks and cache timing attacks. The security monitor manages enclave creation and destruction, enclave state transitions, and interrupt handling, and enclaves are restricted

from performing system calls or I/O. Sanctum modifies the MMU with two Page Table Base Registers (PTBRs) that only the security monitor can change and ensures that only certain memory pages can be referenced by enclave page tables which are verified during initialization.

2.1.5 AEGIS

AEGIS [179] is a Tamper-Evident Environment (TE), and one of the oldest architectures, that detects any memory tampering caused by software or hardware modifications. The Private and Authenticated Tamper-Resistant Environments (PTRs) provided by AEGIS offer even stronger guarantees and protect the confidentiality of security-critical source code and data. The CPU die is considered a trusted component, while DRAM and other peripherals are not considered part of the Trusted Computing Base (TCB). AEGIS executes legacy code and its protection mechanisms are enabled by calling the `enter_aegis` instruction which isolates the program and detects any memory tampering. AEGIS utilizes a hash-tree data structure to validate data integrity when data chunks are read into the on-chip caches. Also, the system provides remote attestation triggered by the `sign_msg` instruction, which hashes the provided data along with the protected process's hash and signs the result with a CPU-specific private key, asymmetrically. AEGIS can be implemented in hardware or software utilizing a Security Kernel (SK), and in PTR mode, the CPU guarantees confidentiality by encrypting the blocks with AES-CBC using 32-bit random initialization vectors (IVs).

2.1.6 Apple Secure Enclave

The Apple Secure Enclave [38] is a dedicated, secure subsystem integrated into the company's A-series chips that provides an isolated execution environment for sensitive operations on Apple devices. The Secure Enclave is independent of the main CPU and has its own isolated memory and processing resources. It uses a unique ID and secure boot process to establish a chain of trust that verifies the integrity of the device's software, ensuring that only authorized code can access sensitive data. The Secure Enclave is used for several security features on Apple devices, including biometric authentication via Touch ID and Face ID, secure payments using Apple Pay, and encrypted data storage. The Secure Enclave also provides a secure environment for executing sensitive code and data, protecting against both software- and hardware-based attacks. The system is designed with multiple layers of security features, including hardware-based encryption, anti-replay and anti-tamper protection, and a dedicated hardware random number generator. Additionally, the Secure Enclave cannot be accessed by the main CPU or other components, further reducing the attack surface.

2.1.7 Bastion

Bastion [57] is a system that combines a hardware and software architecture to ensure confidentiality and integrity for security-critical software modules. It relies on a trusted hypervisor and a modified processor that provides memory protection to guard against physical attacks but only single-core processors are currently supported. Bastion aims to mainly protect the hypervisor since everything apart from the microprocessor and the hypervisor is considered untrusted. The secure launch computes a cryptographic hash over the code and data of the hypervisor and stores the result in a CPU register. The implementation of secure launch and the register contents are not modifiable via software. After the hypervisor is loaded, the software modules can invoke a new secure launch hypercall for initialization purposes. The modified CPU ensures that the hypervisor is invoked for every TLB miss and is responsible for checking whether the virtual address responsible for the access corresponds to the one associated with the physical page and a specific software module. All untrusted software not belonging to a specific module is treated as a module with a zero ID, ensuring that untrusted software cannot access code or data belonging to protected modules. The hypervisor saves all the required state information, such as register contents, and wipes all sensitive data before calling the interrupt handler.

2.1.8 Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) [93] is a coprocessor located on the motherboard that can store cryptographic keys and perform attestation. The TPM interacts with software explicitly and provides limited protection against physical attacks. The TPM is equipped with an Endorsement Key (EK) that serves as a master key for all operations provided by the TPM, Attestation Identity Keys (AIKs), and storage keys. The TPM contains Platform Configuration Registers (PCRs) that can store successive hash values for code or data sent to the TPM and are important for remote attestation. The minimum functionality that a TPM can perform is divided into three operations: (i) binding code or data to a given device, (ii) attesting to another party that the device is currently running a certain software configuration, and (iii) sealing code or data to a given device in a certain software configuration. Attestation and sealing only work correctly if the platform configuration is measured from the earliest boot step up to the currently running software component. The biggest disadvantage of the standalone TPM is its restriction in excluding malicious software from the measurement.

2.1.9 AMD PSP

Unlike Intel's CPUs that employ Platform Trust Technology (PTT), AMD processors use a Platform Security Processor (PSP), also referred to as AMD Secure Technology. This sys-

tem is a trusted runtime environment that was integrated into AMD processors around 2013. The PSP is responsible for several tasks including managing the boot process, initializing various security-related mechanisms, monitoring the system for suspicious activity, and implementing appropriate responses. Essentially, the PSP is an ARM kernel with the TrustZone extension integrated into the main CPU as a coprocessor. The PSP firmware is signed by AMD and redistributed via UEFI image files. It runs before the main CPU and the firmware boot process begins right before the basic UEFI loads. The firmware runs within the same system memory space as user applications and has unrestricted access to MMIO. For practical purposes, the difference between AMD's PSP and Intel's PTT is negligible as both comply with the TPM security protocol.

2.1.10 Intel Management Engine (ME)

The Intel Management Engine (ME) [156] is an autonomous subsystem embedded in all Intel processors since 2008. It is used to support the Intel Active Management Technology (AMT) and has recently been used as a Trusted Execution Environment (TEE) for executing security critical processes. The ME is composed of a processor, cryptography engine, Direct Memory Access (DMA) engine, Host Embedded Communication Interface (HECI) engine, Read-Only Memory (ROM), internal Static Random Access Memory (SRAM), timer, and other I/O devices. It always runs as long as the motherboard is receiving power, even when the host system is turned off. The ME runs on the Intel Quark x86-based 32-bit CPU and the MINIX 3 operating system, with its state stored in a partition of the SPI flash using the Embedded Flash File System (EFFS). The ME has its own MAC and IP address for the out-of-band interface with direct access to the Ethernet controller and communicates with the host via the PCI interface. The ME also uses a reserved DRAM region on the host which is dedicated to the ME and is not accessible by the operating system.

2.1.11 x86 System Management Mode (SMM)

The x86 System Management Mode (SMM) [66] is an execution mode similar to *Real* and *Protected* modes available on x86 platforms. SMM provides a hardware-assisted isolated execution environment for implementing platform-specific system control functions, such as power management, and it is initialized by the BIOS. SMM is triggered by asserting the System Management Interrupt (SMI) pin on the CPU. This pin can be asserted in a variety of ways, such as writing to a hardware port or generating message signaled interrupts with a PCI device. Once triggered, the CPU saves its state to a special memory region called System Management RAM (SMRAM) and then atomically executes the SMI handler, stored in SMRAM by the BIOS during boot time. The SMRAM cannot be addressed by the other modes of execution and by default the requests for SMRAM addresses are forwarded to video memory. This design property allows the SMRAM to be used as a secure storage lo-

cation. Moreover, the SMI handler has unrestricted access to the physical address space and can run privileged instructions; thus, SMM is often referred to as Ring -2. Exiting the SMM and resuming to the previous mode is done via executing the `RSM` instruction.

2.2 Further Research on TEEs

The AMD Secure Memory Encryption (SME) [105, 106] is a security feature that addresses physical access attacks. It utilizes a randomly generated encryption key, using the AMD secure processor. The key is loaded into the memory controller at boot time to encrypt the memory. The OS is able to leverage the SME by setting the encrypted bit (C-bit) in the x86 page table. When the C-bit is set, access to that memory page is directed to the AMD Memory Encryption Engine. In the SME design, all devices can access the encrypted memory pages through DMA. The Transparent Secure Memory Encryption (TSME) [105, 106] is a hardware security feature in which all memory pages are encrypted transparently at boot time. This feature is enabled through a BIOS setting. This encrypted memory is transparent to the underlying OS and user software.

Sancus [134] is a hardware-only Protected Module Architecture (PMA), designed by Noorman et al., for lightweight embedded devices such as wireless sensor nodes. In addition to isolating an application's code and sensitive data, it also provides support for remote attestation. It adds secure linking functionality as well, enabling applications to verify modules that they depend on. Soteria [91] is an extension of Sancus which takes advantage of the architecture's functionality to add code confidentiality. Brasser et al. propose TyTan [54], an architecture for lightweight devices which provides isolation between tasks, secure IPC with sender and receiver authentication and real-time guarantees. TyTan's TCB consists of both hardware and software components. TrustLite [111] is a generic PMA for low-cost embedded systems, developed by the Intel Collaborative Research Institute for Secure Computing. In TrustLine, a *trustlet* isolates software components, providing confidentiality and integrity guarantees for both its code and data. The architecture provides OS-independent isolation of *trustlets*, attestation of *trustlets*, trusted inter-process communication, secure peripherals, and interrupt support. The system was implemented as an extension to the Intel Siskiyou Peak research platform. One of the first designs to use hardware and software co-design to built a lightweight trust architecture is SMART [77], proposed by El Deffawy et al. SMART establishes a DRoT in remote embedded devices, providing a minimal architecture that only requires the smallest possible set of hardware changes to implement remote attestation. SMART prototypes that demonstrate the feasibility of the design are built on open-source clones of the ATmega103 and openMSP430.

SecureBlue++ [53, 206] is PMA proposed by IBM which isolates Secure Executables (SEs) from each other and protects the confidentiality and integrity of their data and code. The main architectural changes involve a Memory Protection Unit (MPU), using differ-

ent mechanisms at each level of the memory hierarchy. It also protects against physical memory attacks as well as the introduction of new instructions. Iso-X [79] is an isolated execution architecture where memory can be assigned dynamically to Untrusted and Trusted Partitions which contain compartments. These compartments are essentially protected modules and a developer can indicate which parts of the code should be compartmentalized. The architecture also includes a remote attestation mechanism in hardware, based on asymmetric signatures. Intel also proposed the Trusted Execution Technology (TXT) [92], aiming to overcome the restriction of all software having to be part of the TCB when relying on the TPM. Intel TXT uses the TPM chip but allows for the dynamic establishment of a new Root of Trust (RoT) for software running in a virtualised environment, besides the usual software stack.

2.2.1 GPU-assisted TEEs

Vasiliadis et al. proposed PixelVault [192], a GPU-assisted TEE developed using NVIDIA GPUs, able to store cryptographic keys and carry out cryptographic operations exclusively on the GPU. The system can protect the secret keys from leakage even in cases where the host becomes completely compromised. This is achieved by exposing secret keys only in GPU registers, keeping PixelVault's critical code in the GPU instruction cache and preventing any access to both of them from the host, leveraging the non-preemptive execution model of the external GPUs. Cook et al. [65] presented a mechanism to transfer encrypted video that is only decrypted once on the GPU. In 2018, Volos et al. presented Graviton [197], a system that enables applications to offload security-sensitive and performance-critical GPGPU kernels and data to a GPU. Graviton enables GPGPU kernel execution in isolation from other code running on the GPU and all software on the host, including the device driver, the operating system, and the hypervisor. Graviton can be integrated into existing GPUs with relatively low hardware complexity and is one of the most complete GPU-based TEEs. DeepAttest [59] is the first on-device DNN attestation framework that verifies the legitimacy of the deployed DNN before allowing it to execute normal inference. While DeepAttest is not a TEE system on its own, it extends and utilizes Graviton to offer the proposed functionality. Yu et al. [211] introduced a system that relies on a privileged host component to enforce isolation between virtual machines and display. Jang et al. proposed a hardware/software architecture, called HIX (Heterogeneous Isolated eXecution), that aims to isolate GPUs even from possibly malicious high-privileged software, such as the OS or hypervisor, by modifying the I/O interconnect between the CPU and GPU. Finally, Telekine [99] is a system that provides secure access to cloud-based GPUs using a data-oblivious communication between the client and the GPU-based TEE.

2.2.2 FPGA-assisted TEEs

Recently, FPGAs have become a popular tool for designing and implementing trusted execution environments. Pereira et al. [146] proposed a new TEE designed, named Trusted Execution Environments On-Demand (TEEOD), that utilizes FPGAs to dynamically provide secure execution environments for security-critical applications with high-bandwidth connections, physical on-chip isolation and configurable software and hardware TCBs. MeetGo [139] is an FPGA-based TEE, targeting remote computing, that operates independently of the host system. The system offers a remote attestation mechanism to verify the integrity of the applications implemented as hardware logic, an isolation mechanism that prevents unauthorized CPU access to these applications, and a secure communication mechanism for protected transmissions of sensitive data between the applications and remote users. SGX-FPGA [208] is a system that provides a trusted hardware isolation path, aiming to bridge SGX enclaves and FPGAs in heterogeneous x86 CPU/FPGA architectures, and protects data either stored in both devices or in transmission. On the other hand, Ambassy [101] targets hybrid ARM/FPGA architectures with a framework that enables secondary TEEs and two-way sandboxing of the new TEE running third-party TAs.

BOYTee [40] is a novel framework that enables users to build general-purpose enclaves with configurable hardware and software TCBs while the authors demonstrate the system and its toolchain on the Xilinx SoC FPGA. More recently, Zhao et al. introduced ShEF [218], a TEE that targets cloud-based reconfigurable accelerators. ShEF provides a secure boot and remote attestation process that relies solely on existing FPGA mechanisms for RoT, as well as a component for secure data access on the accelerator. Zhu et al. [222] introduced a heterogeneous TEE, called HETEE, that leverages the PCIe ExpressFabric to allocate accelerators to a server node on the same rack during non security-critical tasks, and allocates them back into a secure enclave in response to requests for confidential computing. Finally, Coughlin et al. [69] proposed a mechanism, based on a Xilinx Zynq UltraScale+ FPGA, that removes the RoT dependency to third-party processes, offering a self-provisioning stage for key generation and a secure update mechanism.

Chapter 3
Security Framework Overview

In this work, we design, implement, and evaluate a modular TEE-based framework, composed of interoperable components, that aims to address the security issues of key software layers found in modern desktop and mobile systems. Our goal is to raise the bar in the areas of (i) *host protection*, (ii) *OS service protection*, (iii) *secure communications*, and (iv) *secure dynamic code execution*. Also, we aim to prove that enclave-based trusted execution environments can serve as a readily available common platform to base such a framework on, enabling the development of the required security modules as well as the protection of the framework itself.

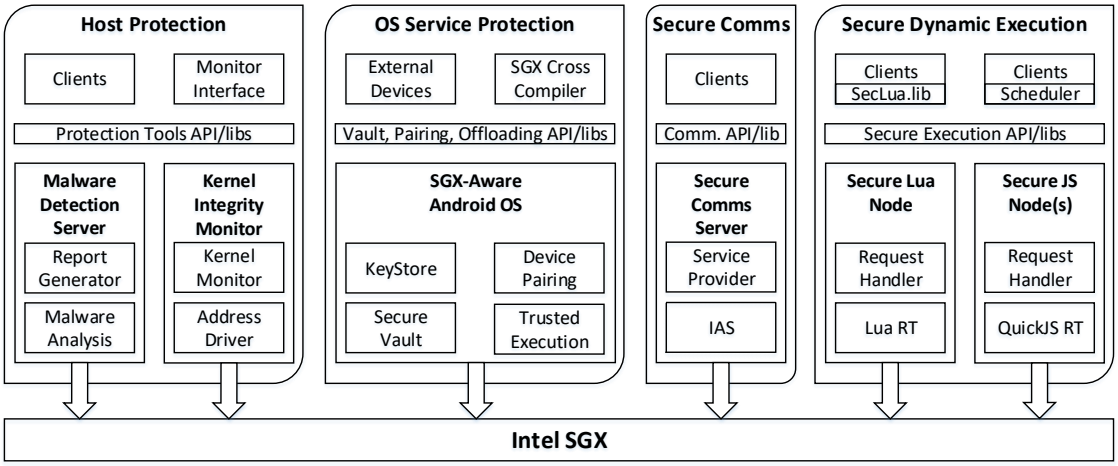


Figure 3.1: Enclave-based security framework overview.

In this chapter, we briefly present the six interoperable components that compose our TEE-based security framework. The system provides two attack detection systems (malware scanning and kernel monitoring), an enclave-enhanced mobile OS, a secure CPU-to-CPU communication system and two secure runtimes with one of them providing scale-out capabilities. One or more components can simultaneously co-exist on the same host,

sharing the common SGX infrastructure, and can be utilized through our provided clients, libraries and APIs. An overview of the framework's architecture is presented in Figure 3.1.

3.1 Host Protection

To cover the area of host protection against malware and rootkits, our framework provides two distinct components: (i) a *malware detection* solution, named TrustAV, and (ii) a *kernel integrity monitor*, named SGX-Mon.

TrustAV is a cloud-based malware detection system that utilizes the underlying Intel SGX enclaves for privacy and security. The system protects user data by offloading malware detection to a remote SGX-enabled server, ensuring privacy even in untrusted environments. The detection engine relies on binary signatures and regular expressions and provides client applications for various devices. The architecture aims to address the performance constraints of Intel SGX and provides a caching scheme to reduce the required protected memory footprint.

SGX-Mon is a kernel integrity monitor that aims to ensure the operating system's kernel integrity. The monitor resides in the user space, utilizing the trusted execution environment to avoid detection and tampering. This component scans, analyzes, and verifies critical kernel memory pages and regions for integrity. The system periodically compares the contents of these regions against known benign values. Kernel memory locations can be acquired during a secure bootstrap phase and are mapped into enclave memory using our custom driver. The system also provides attestable heartbeats, monitors potential adversary mappings, and operates alongside the rest of the framework's components.

3.2 Operating System Service Protection

To provide trusted execution capabilities to mobile and low-power devices, we extend the framework with our SGX-enabled Android OS, named Andromeda. This component is an enhanced Android operating system with trusted execution capabilities, achieved by porting the SGX framework to Android. This includes the necessary drivers, libraries, services, and APIs, allowing developers to utilize SGX for their mobile applications. The enhanced Android OS supports the installation and execution of SGX-based applications on compatible devices without modifications. Andromeda also improves popular Android services by incorporating enclaves, enhancing security and enabling secure management of cryptographic keys and sensitive data. These enhancements are transparent to existing Android APKs, maintaining forward and backward compatibility. Additionally, this component enables the execution of multiple enclaves simultaneously, providing isolated secure environments on a per-application or per-function basis. Furthermore, Andromeda facilitates trusted data flow control between externally paired devices and the SGX-enhanced

Android host, enabling secure offloading of data storage and computations without requiring TEE-enabled CPUs on the external devices.

3.3 Secure Communication

Our framework's fourth component targets the area of secure and attested communications. This system provides secure communication between SGX-enabled applications, even when one end is potentially located in an untrusted remote environment. The module utilizes Intel's Remote Attestation process for secure key exchange and attestation. Also, it offers a simple API for entities to verify and attest each other, creating an encrypted SGX-to-SGX (CPU-to-CPU) communication channel. The system implements an optional caching system to reduce connection latency and can operate either as a stand-alone component or serve as the default communication agent between the rest of the framework's modules, when applicable. Our comparative analysis conducted against commonly used secure communication methods, such as TLS, showcase its performance and practicality.

3.4 Secure Dynamic Execution

The last two components of our security framework focus on the area of secure dynamic code execution and provide transparent trusted execution for dynamic languages, such as Lua and JavaScript, without requiring modifications of legacy code, knowledge of the underlying TEE or explicit code partitioning. Both systems are based on language interpreters implemented in C/C++, which we port into enclaves, that serve as the static and protected environments where code can be loaded as data and executed securely.

The first system, named LuaGuardia, is developed to provide secure execution of new and legacy applications developed in Lua. The secure execution functionality is offered as a service, either locally on the target device, or remotely, allowing developers to run existing or new applications without the need to develop enclave code and link it via custom bindings. This component also enables embedded and low-power devices that are not equipped with enclave capabilities to execute code securely via offloading. Additionally, LuaGuardia's runtime provides optional tags that can be used to designate specific code blocks or functions for secure execution, enabling fine-grained secure partitioning.

Our framework's final component, named Atlas, is designed and built based on the same principles that led to the development of LuaGuardia but targets the secure execution of JavaScript code. Atlas also extends LuaGuardia's model with a scheduling system for scaling out components on TEEs to facilitate secure distributed applications. This component automatically offloads function calls and distributes the load among secure nodes, ensuring minimal developer effort. Splitability annotations and code analysis are also utilized to confirm compatibility with the original program's semantics.

3.5 Threat Model and Assumptions

In this work, we assume powerful and active adversaries who possibly have root privileges and access to the physical hardware hosting our framework's component(s). The adversaries may be able to control the entire software stack, including the OS kernel and other system software. However, we explicitly exclude denial-of-service (DoS) attacks on enclaves, given that the design of SGX allows the host OS to control an enclave's life cycle anyway. As a result, an attacker can prevent or abort the execution of enclaves, but should not gain any knowledge by doing so. Moreover, side-channel attacks that exploit timing or page faults, or attacks based on vulnerabilities of the application running inside the enclave are proven to be feasible on SGX enclaves. However, protecting SGX enclaves from side-channel attacks that either focus on software or hardware bugs is orthogonal to our work and thus we consider them to be out of the scope of this work. However, any successful attempt to protect SGX-enabled software/hardware has a direct benefit to our framework. Finally, we assume the design and implementation of SGX itself (including all cryptographic operations, the SGX SDK and PSW, the driver, and the required SGX services) is secure and does not contain any vulnerabilities.

When client applications or client libraries are involved, we assume that they may be distributed over different networks and geographical areas. These nodes are connected over a public, not necessarily trusted, network, over which they are reachable and able to transmit and exchange data. We also assume that the SGX-enabled client nodes may be compromised by a powerful adversary with full-privileged access or even access to the physical hardware. All client-side enclaves running on SGX hardware-mode are considered part of our TCB, as well as the Intel SGX Attestation Service that verifies that the SGX hardware is genuine. We also assume that client software is installed and initially executed when the devices are in a clean state, so no malicious executable has taken the control of the client or the device prior to the execution of our components. Finally, we assume that a secure bootstrap phase is available in the context of kernel integrity monitoring.

Chapter 4
Privacy-preserving Malware Analysis

In this chapter, we present the first component of our security stack, a cloud-based malware detection solution designed for a plethora of device types, named TrustAV. The system is able to offload the processing of malware analysis to a remote server, where it is executed entirely inside hardware-assisted secure enclaves. By doing so, TrustAV is capable to shield the transfer and processing of user data even in untrusted environments with tolerable performance overheads, ensuring that private user data are never exposed to malicious entities or honest-but-curious providers. TrustAV also utilizes various techniques to overcome performance overheads introduced by the Intel SGX technology and reduce the required enclave memory — a limiting factor for signature-based malware analysis executed in secure enclave environments — offering up to 3x improved performance compared to its unoptimized implementation.

4.1 Signature-based Malware Detection

Our malware detection engine is based on signature scanning, a commonly used technique in state-of-the-art antivirus systems. The data under malware analysis are processed against a set of malware signatures to identify the presence of malicious software. The Aho-Corasick algorithm [32] is considered an efficient option for multiple pattern searching, since it matches all signatures in a ruleset simultaneously. For this reason, Aho-Corasick is utilised by popular open source security solutions, such as the ClamAV antivirus [6] and the Snort network intrusion detection system [25]). The algorithm constructs a finite state machine that resembles a tree, along with *failure* links between the nodes. Failure links are followed when there is no matching transition, allowing fast transitions to other branches of the tree with a shared prefix, avoiding costly backtracking. To provide a more efficient approach, a Deterministic Finite Automaton (DFA) can be built by unrolling the failure links in advance and adding relevant transitions to map each failure directly to a node without the need to follow multiple failure links at runtime. This property enables us to represent the DFA as a single-dimensional integer array with greatly

decreased memory footprint, compared to representing the automaton as a tree, and implement the DFA traversal function using simple arithmetic operations. Leveraging this serialized DFA, we can achieve malware detection entirely inside the SGX enclaves with a small code and memory footprint.

4.2 Threat Model

Providing security applications as a service (AAS) has become a very convenient trend due to lower cost and maintenance complexity. Still, these workloads contain important information about the end-user(s). More specifically, a malware detection tool has privileged access among the user files, e-mails and network traffic. Processing such sensitive information needs to be taken seriously by complying to security and privacy-preserving standards to guarantee confidentiality.

To specify the treat model for this component, where user data are not handled by the users themselves, we define three different entities: (i) the TrustAV *client*, (ii) the TrustAV *server*, and (iii) the *cloud provider*. The malware scanning engine lives inside the TrustAV cloud-based server, which communicates with the clients through a network connection. We assume that a TrustAV client is installed and initially executed when the device is in a clean state, so no malicious executable has taken the control of the client or the device prior to the execution of our malware detection system. The environment that hosts the TrustAV server is considered untrusted, since there is no control over the operating system, the hypervisor, the drivers, the management stack, the system's memory, I/O devices, etc. Furthermore, even in a fully healthy environment, there is always the possibility of an honest-but-curious cloud provider, willing to learn and extract information regarding the users or the system utilization. In this work, we safeguard both users and the TrustAV system from the aforementioned conditions. Obviously, the client and the server are required to safeguard the transmission and the processing of the user's data. For the purpose of completeness, we assume an uncompromised Intel SGX-enabled processor hosting the remote server. Finally, we stress that handling any side-channel attacks against Intel SGX or software flaws in SGX's implementation is out of our scope.

4.3 Design

In this section, we describe the design and implementation of the TrustAV architecture. The overview of TrustAV is presented in Figure 4.1. The two entities that compose our system are: (i) the TrustAV *client*, which transmits the necessary files to the remote server for scanning, and (ii) the TrustAV *server*, which is responsible for performing the malware detection in a privacy-preserving way. The entire malware scanning operation is performed on the cloud-based server, encapsulated inside Intel SGX enclaves. This encapsulation

enables the protection of the data processing algorithms, the signature set, and most importantly the privacy of the user's data.

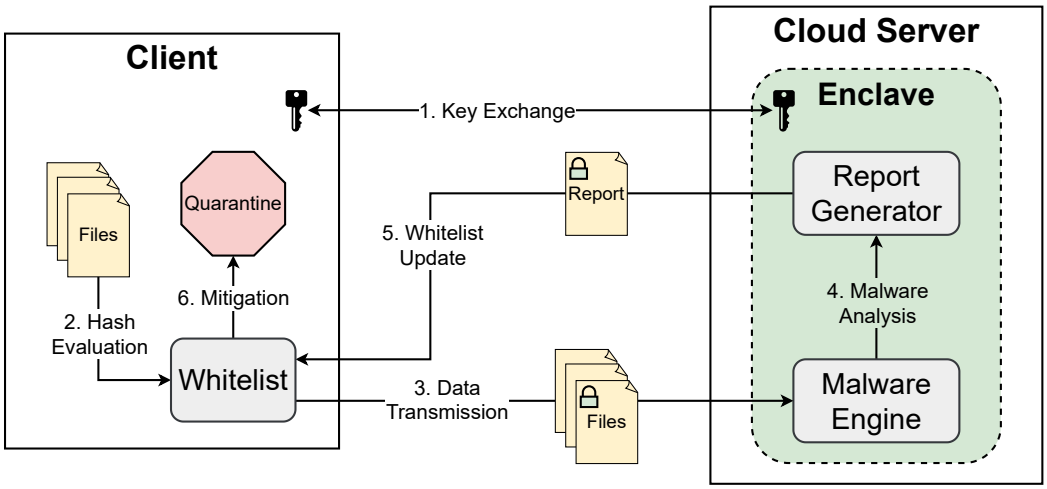


Figure 4.1: Architecture overview of the malware detection service.

4.3.1 TrustAV Client

The TrustAV client implements three distinct functionalities, using three different components. The client (i) prompts the users to select files or file system regions that they wish to scan and provides a set of actions for each file that is found to be infected. Once the selection is defined, (ii) it gathers the data and periodically checks their status by comparing their hash values against known benign hash values kept in a whitelist. Finally, the client (iii) transfers the files whose hash values did not match with the whitelist to the cloud-based TrustAV server which performs the malware scanning and reports the results.

The client is required to provide minimum effort and functionality, while the computationally intensive malware scanning is performed by the TrustAV cloud-based server. In this way, the server remains independent from the client implementation, while multiple client versions can be developed to support different platforms and operating systems.

The hashing component is responsible for retrieving the data from the client's file system. The data can be found in the form of various files present in the file system, such as photos, videos, audio files or applications. The client periodically hashes all selected files and directories and forwards the hash values to the whitelist component. The window of the periodic scanning can be defined by each user, depending on the device type and its current status. For example, the user can opt for a larger window when the system is running on battery in power-saving mode to avoid exhausting the device's resources.

The client's whitelist component is responsible for comparing the hash values obtained by the hash computation module against a list of hashes that have been calculated from a

known clean state of each file. If the hash values do not match a given file, it is marked as suspicious and the client has to forward it to the cloud-based TrustAV server for malware scanning. Moreover, the client is responsible for the maintenance of the whitelist by managing the entries of new or deleted files and updating existing entries with new benign hash values. The module flags each file whose hash value does not match with its corresponding value in the list as potentially infected and thus dangerous. Once this process is finished, all the marked files are forwarded to the Data I/O module to be transmitted to the TrustAV remote server.

The motivation for this periodic hash-checking functionality is threefold. First, calculating and comparing hash values against a set of known hashes — which represent the clean state of files — can be quick and efficient given the plethora of different hash algorithms available. Second, it is a fast preliminary way to filter out benign files from possibly infected ones without having to perform complex malware analysis on every file and application of a device. This allows the TrustAV client to be easily implemented for a wide variety of commodity devices (e.g., desktops, smartphones, IoT devices, etc.). Third, by marking only a limited subset of files as potentially infected, we minimize the amount of data that need to be transferred to the remote TrustAV server, improving the overall performance of our system and minimizing the cost for users who perform virus scanning using metered connections, mobile data plans or are connected via a low bandwidth channel. Also, our microbenchmarks indicated that using the whitelist mechanism improves power consumption for battery-powered devices.

The most important component of the TrustAV client is the secure communication with the remote server. This component ensures that all the potentially infected files are transmitted to the TrustAV remote server for a thorough malware analysis, in a secure and privacy-preserving manner. Each file is first encrypted using a secret cryptographic key, that is established with the server. After the successful transmission of the marked files, it awaits for the remote server's response. This response is received in an encrypted format and contains details about the infected files, such as the risk level and the actions to be taken by the user. According to the information provided by the TrustAV server, the client prompts the user with possible actions to handle the infected files in the file system. Then, the whitelist module updates the list for every file whose contents changed in a benign way, thus leading to the generation of a new hash value, and removes the entries of the deleted files. When applicable, Remote Attestation can be leveraged by the client to further ensure that the contacted TrustAV server operates on genuine SGX-enabled hardware.

4.3.2 TrustAV Server

The second entity of our malware detection system is the TrustAV server, composed of three distinct modules. The server is able to accept connections from multiple TrustAV

clients and perform malware detection on the incoming data. Also, it can be hosted in private or even public cloud infrastructure, or on a dedicated host.

The server maintains an updated signature set, used for the malware analysis. By keeping the entire signature set on the remote server, the system is able to benefit in two ways. First, the detection does not rely on each user to maintain the latest ruleset locally, on the client device. The local update process could often be neglected by a large number of users, thus allowing the latest malware to operate undetected on multiple devices. Moreover, the users do not have to store the signature set on their device, typically sacrificing valuable storage space in case the device is a smartphone, tablet or an IoT device with limited storage. Second, a major benefit of offloading the entire malware analysis on the cloud-based TrustAV server is the ability to utilize Intel SGX enclaves, even when not supported by the client device. Using SGX enclaves we are able to execute the entire life cycle of the virus scanning process in a trusted environment, ensuring that any sensitive data obtained by the users, cryptographic keys and malware signatures are never exposed in the server's DRAM or file system. This attribute is crucial for two reasons. First, we can guarantee that users can securely offload sensitive data to the remote server for malware analysis without risking leakages. Second, even if malicious actors manage to compromise the server, they are not be able to identify the signatures used in the ruleset or tamper them in any way. Moreover, Intel SGX enclaves ensure the secure execution of the malware detection engine. This makes the virus scanning algorithms immune to attacks while they are executed, preventing code tampering or data leakage from variables in use. Finally, since enclaves operate as reverse sandboxes and the enclave hosting the malware scanning engine only communicates with the client, user data are not accessible even by honest-but-curious providers hosting the server, ensuring the privacy of the offloaded user data. Any attempt to read or tamper SGX-protected memory during the server's execution is prevented by the runtime.

User data are received in an encrypted format by the TrustAV server and are forwarded inside the Intel SGX enclave hosting our system's engine. Once inside the secure enclave, the data are decrypted and prepared for processing. The cryptographic keys required for the successful decryption of the client's data exclusively reside inside the SGX enclave. In this way, the secret keys and sensitive data, such as personal documents or photos, are never present in plain-text format in the server's file system or DRAM and they remain inaccessible even by the server's host/provider. Moreover, even if the non-SGX part of the TrustAV server or the hosting infrastructure gets compromised, the keys, malware signatures and private user data cannot be obtained.

The malware scanning component constitutes the core of our system and is responsible for processing the incoming data using a given malware signature set. The entire functionality of this component along with all required data, such as the ruleset, reside only inside the Intel SGX enclave. Each rule is assigned with a unique ID and consists of a set

of patterns and information metadata, describing the malware functionality, the risk level and suggested actions. The virus scanning process is performed inside the enclave once the data are decrypted using each client's secret key. When a rule is successfully triggered by an input file, the corresponding metadata along with the file information are processed to be forwarded to the client as a status report.

The report generation is performed by a separate component, which also resides inside the secure enclave. During this process, the TrustAV server receives results from the malware scanning engine and generates a report that will be processed by the TrustAV client. This report contains information about the malicious files detected, such as filenames, risk level, scanning date, etc. When the report is constructed, the server encrypts it, while still inside the enclave, and then it is forwarded to the corresponding client. Ensuring that the scanning report never lives outside of the SGX enclaves in plain-text format is very important for the user's privacy. By rendering the report inaccessible outside the user's device or the server's SGX enclave, attackers or honest-but-curious entities, such as the service provider, cannot obtain any information about the user's private data. In combination with protecting the signature set inside the enclaves, we also eliminate the possibility of malicious entities injecting custom signatures and observe the generated report to infer information that could threaten the privacy of the user's data. Moreover, the report is randomly obfuscated so that its size cannot be used to infer information about the number or type of identified patterns.

4.3.3 Service Registration

The registration process is the first task performed by TrustAV when the client is initiated on a user's device. At the first step of this process, the client communicates with the TrustAV cloud-based server and establishes a shared key. During this process, the server generates a client ID and stores the shared key along with the corresponding ID inside the SGX enclave. The second step is the generation of the list containing the hashes of each file at a clean, uninfected state. For this reason, the client hashes every selected file and temporarily populates the whitelist. Then, every file is transmitted to the remote server for malware analysis. Once the server responds with the first report, the hash values of the uninfected files are considered permanent in the whitelist and all the malicious files, if any, are handled by the user according to the report's suggestions. When the registration process is finished, the TrustAV client prompts the user for a periodic hashing interval and the system defaults in automated periodic scanning.

4.3.4 Remote Attestation

TrustAV can leverage the Remote Attestation services, provided by Intel, to further increase the security and level of trust of the SGX-enabled server. Using remote attestation, the

TrustAV client challenges the server to verify that the core part of the engine is located inside a signed SGX enclave, executed on an SGX-enabled processor in *hardware mode*. In this way, we eliminate the possibility of a malicious entity posing as an SGX-enabled TrustAV server to obtain access to users' private data. Moreover, we prevent entities executing the TrustAV server in SGX *debug* or *simulation mode*, trying to obtain access to the user's sensitive data and the server's secret keys using debuggers.

4.4 Implementation

In this section, we present the implementation of the applications that compose TrustAV. We provide detailed description of the malware detection process as well as the steps required to develop an SGX-enabled malware scanning engine, operating on a cloud-based server to protect and preserve the privacy of the user's data.

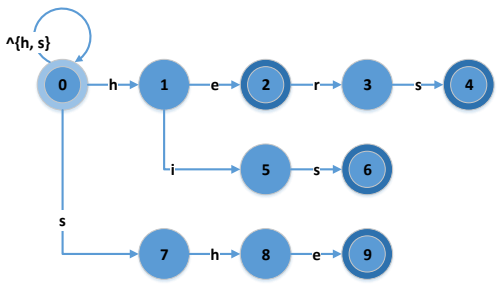
4.4.1 Malware Scanning Inside SGX Enclaves

Our malware scanning engine is based on the Aho-Corasick pattern matching algorithm, as it is one of the most efficient and widely used algorithms, found in many signature-based solutions, such as the popular open-source ClamAV antivirus. TrustAV performs the entire malware scanning process inside Intel SGX enclaves to preserve the privacy of the offloaded data, the security of the executed code and the integrity of the signature set.

As described in this chapter's introduction, in most implementations, the patterns are compiled into a state machine (DFA) which is constructed as a tree, with each node containing information about the state it represents, as well as various metadata and pointers to the connected nodes. However, this state machine structure is not optimized to be used inside SGX enclaves. The reason is twofold: (i) the generated tree requires a lot of memory to be represented while the live available enclave memory is quite limited (when SGX2 hardware is not available), (ii) traversing the nodes scattered in memory during the pattern matching process eliminates all caching effects and reduces the sustainable performance.

To address these constraints that can play a significant role in terms of performance, we choose to represent the DFA as a serialization of the state machine tree to a single-dimensional integer array. We describe the serialization process using a two-dimensional array (for better visualization) and the tree presented in Figure 4.2(a), produced by the patterns *{he, she, his, hers}*, as an example.

First, we compile all the available malware signatures into a single Aho-Corasick DFA. Next, we serialize the produced tree as a two-dimensional integer array. This array consists of 256 columns, which represent the size of the American Standard Code for Information Interchange (ASCII) set (i.e., all the values that a single input byte can take), and N rows, where N is the number of the states in the DFA. Each row represents a DFA state and each



(a) Aho-Corasick tree.

ASCII set

States	0	...	e 101	...	h 104	i 105	...	r 114	s 115	...	255
0	0	0	0	0	1	0	0	0	7	0	0
1	0	0	-2	0	1	5	0	0	7	0	0
2	0	0	0	0	1	0	0	3	7	0	0
3	0	0	0	0	1	0	0	0	-4	0	0
4	0	0	0	0	8	0	0	0	7	0	0
5	0	0	0	0	1	0	0	0	-6	0	0
6	0	0	0	0	8	0	0	0	7	0	0
7	0	0	0	0	8	0	0	0	7	0	0
8	0	0	-9	0	1	5	0	0	7	0	0
9	0	0	0	0	1	0	0	3	7	0	0

(b) Aho-Corasick DFA serialized as an integer array.

Figure 4.2: An overview of the Aho-Corasick DFA serialized as an integer array. The DFA contains the patterns *{he, she, his, hers}*. In sub-figure (a), dark blue nodes indicate final states (i.e., end of a pattern) while the same information is indicated with negative values in sub-figure (b).

cell contains the ID of the next valid transition, corresponding to the ASCII character that the cell represents. An example of this array can be found in Figure 4.2(b).

To traverse the serialized DFA tree, the matching process starts from state 0 (row 0) and selects the appropriate column, according to the ASCII value of the first character of the input. In this cell, it finds the next valid state, which is located in another row of the array. Then, it fetches the next character from the input and moves to the cell pointed to by the row given in the previous step and the column given by the ASCII value of the current character. The final states in the array are annotated with a negative sign to reduce memory utilization. When the task encounters a negative state, a match has been successfully found. Then, the search is continued using its absolute value for the next step. The fail states either point the matcher to a previous valid state or to the initial state 0. In practice, as we stated earlier, this array is single-dimensional and all the rows mentioned in our example are concatenated. Since the size of every row is 256 integers, the matcher traverses the array as follows: *state = dfa[state * 256 + current_input_byte]*. For this reason, the serialization of the state machine provides a second important benefit to our system. The malware scanning function requires only a few lines of code, rendering it very fast and efficient but, most importantly, produces a minimal Trusted Computing Base (TCB) which is very easy to audit and eliminate any security threatening software bugs.

4.4.2 SGX Enclave I/O

With the malware scanning process and the signature set secured inside SGX enclaves, the second, and most important, part of our implementation is to provide secure and efficient I/O with the enclave. The only entry point offered by the enclave is utilized for user data

entry. Since the SGX enclaves do not have access to system calls, the network sockets, necessary for receiving client data, are managed by the non-SGX enabled part of the TrustAV server, as SGX's model dictates. The data arrive encrypted over the network while the secret keys required for their decryption exclusively reside inside the SGX enclave. Entering the enclave is achieved using an `ECALL` function. However, multiple consecutive calls to this function can impose a performance overhead to the system. For this reason, we bach incoming encrypted client data using a buffer in the non-SGX enabled part of the application and transfer the buffer into the enclave once it is full. The size of this buffer can be optimized dynamically according to the current workload. Once a batch of user data is gathered, it is forwarded into the enclave for processing.

Before the malware scan can take place, the matcher decrypts the user data with the corresponding key, inside the secure enclave. This ensures that while a compromised server can block data forwarding to the enclave, the data and the secret keys never reside in main memory or the file system in plain-text format. The malware analysis results are compiled as a report that can be then parsed by the client. This report contains information about the infected files and the identified malware, as well as recommended actions that the user could perform to mitigate each threat. The only data output point of the enclave is utilized to transmit the report back to the TrustAV client. Once again, the report is encrypted inside the SGX enclave using the client's secret key. This action is performed to guarantee that external observers are not able to gain any useful information that could disclose the user's file contents or types. After its successful encryption, the report is forwarded to the non-SGX enabled part of the TrustAV server to be transmitted to the client via the network. As an extra privacy-preserving guarantee, the report is generated each time with an arbitrary size (obfuscated) to prevent attackers from inferring information by monitoring the report's size.

4.4.3 Performance Optimizations

One of the biggest challenges of modern signature-based intrusion/malware detection solutions (such as Snort, ClamAV, etc.) is minimizing the memory footprint of the signature automata. Usually, the performance of applications that utilize multi-pattern matching algorithms, such as Aho-Corasick, is limited by the cache size provided by the CPU. Once the signature automaton exceeds the cache size, cache misses can greatly hinder the system's performance — a problem that only gets worse as the automaton increases in size when new rules are added. This challenge reaches a new dimension when such an antivirus engine is designed to be executed inside secure SGX enclaves. During execution, enclave code and the required data are placed in a special memory region called Enclave Page Cache (EPC). This region is protected by being encrypted by a dedicated chip called Memory Encryption Engine (MEE). In this way, memory pages are only decrypted inside

the physical processor core while external reads on the memory bus can only observe encrypted data. The EPC size is set in the BIOS and can reach up to 128MB. The SGX driver for the Linux platform supports page swapping, allowing SGX to remove pages from the EPC and place them encrypted in unprotected memory, as well as restore them when they are referenced. Pages cannot be removed until all cache entries referencing them have been removed from all processor logical cores. This property of the SGX driver provided for Linux allows the allocation of the memory required to store automata exceeding 128MB. However, the random accesses in the automaton's states during the matching process produce a substantial number of accesses to pages stored in unprotected regions, triggering the expensive process of restoring such pages — an overhead that is being added to the one already introduced by the increased number of cache misses, as at this point the automaton size is greater than the cache size by orders of magnitude. During the preliminary testing of our system, we discovered that this issue becomes very prominent when more than 50% of the input data are infected with malware. Similar behaviour deriving from EPC size is also observed in the literature [41]. Moreover, the SGX driver for Windows does not support this page swapping functionality, meaning that automata greater than 128MB cannot be stored inside enclaves at all. Considering that automata containing 5000 patterns exceed 128MB, even after serialization, the EPC size is a strong limiting factor for porting our application to Windows, as well as for signature-based solutions that wish to utilize SGX enclaves with the commonly available SGX1-compatible hardware.

To address this constraint, we develop two custom caching systems, aiming to minimise accesses to pages stored out of EPC as well as enable the protection of big automata for the Windows platform without sacrificing security. Our first approach is a simple cache with configurable size that is limited to 90MB, allowing other data and code to be stored inside the EPC without triggering swapping. This cache stores the first N automaton states while the rest are stored in untrusted memory. To protect this part of the automaton, we encrypt each individual transition separately using AES-GCM as provided by the SGX SDK. This process is performed using SGX enclaves for the entire automaton during its compilation, enabling the reconfiguration of the cache size, while the keys required for decrypting the transitions are exclusively stored in the enclave. The SGX enclave is able to access untrusted memory with minimal overhead. During execution, if a transition is not found in the cache, the matcher fetches a copy of its encrypted counterpart, decrypts the contents inside the enclave and proceeds with the malware scanning. In this way, we eliminate the need of costly secure swapping of EPC pages and enable the use of big protected signature sets for Windows. As we describe in Section 4.5, this caching system is very efficient due to the fact that according to our microbenchmarks, in most cases, caching 25% of the automaton results to an average cache hit ratio of 85% when the data are 100% infected (i.e., all automaton states are utilized), which also is the worst possible case.

Utilizing the same encryption scheme, we further optimize the caching system by re-

placing the simple mechanism with an LRU cache. Our goal at this step is to further improve the cache hit ratio while decreasing the cache's memory footprint to minimise cache misses and possibly enable the simultaneous operation of several caches, serving different ruleset automata. To achieve this, we implement the LRU as a double-linked list with each node holding information about the cached transitions. LRU look-ups are performed using a hash table with each entry being a pointer to a queue node to minimise the memory footprint. Finally, we eliminate constant memory allocations when transitions are inserted or evicted from the cache by implementing a custom memory pool that performs all required allocations during the system's initialization phase.

4.4.4 TrustAV Clients

Aiming to cover the vast majority of available devices and platforms, we implement both desktop and mobile clients. The desktop version is targeted for traditional desktop/server devices while the latter is developed as a standard Android APK that can be utilized by smartphones, tablets and Android-enabled IoT devices. Both clients perform the service registration with the server, as described in Section 4.3. The hashing of the selected files is performed using SHA-256, utilizing the appropriate libraries offered by each platform. Moreover, the clients implement secure persistent storage of the whitelist by exporting it to the file system in an encrypted format. The exported whitelist is also paired with checksums to ensure its integrity. The main difference between the two clients is the fact that the desktop implementation is always able to utilise remote attestation to further enhance the security of the connection with the server. This functionality is not available for the Android platform for every use case but is not a strong requirement for the execution of our system. Finally, the mobile client offers several options for the fine-grained configuration of the scanning intervals to minimise network traffic when the device is using a metered connection and optimise battery consumption. Clients targeting other platforms can be easily developed by implementing the described functionality, using our API.

4.5 Evaluation

In this section, we present TrustAV's evaluation. First, we analyze the performance characteristics of our malware scanning engine and explain the performance overhead introduced by the usage of hardware assisted enclaves (i.e., Intel SGX enclaves). Then, we evaluate the performance of our caching systems and present their effectiveness compared to default SGX secure memory page management.

4.5.1 Evaluation Setup

Hardware Setup

The TrustAV server is hosted on an commodity desktop, based on an eight-core Intel i7-8700K CPU, running at 3.7GHz, providing support for Intel SGX enclaves. The system is also equipped with 32GB of DDR4 RAM clocked at 2400MHz and is connected to the network using a 1 GbE NIC.

Malware Signatures and Workloads

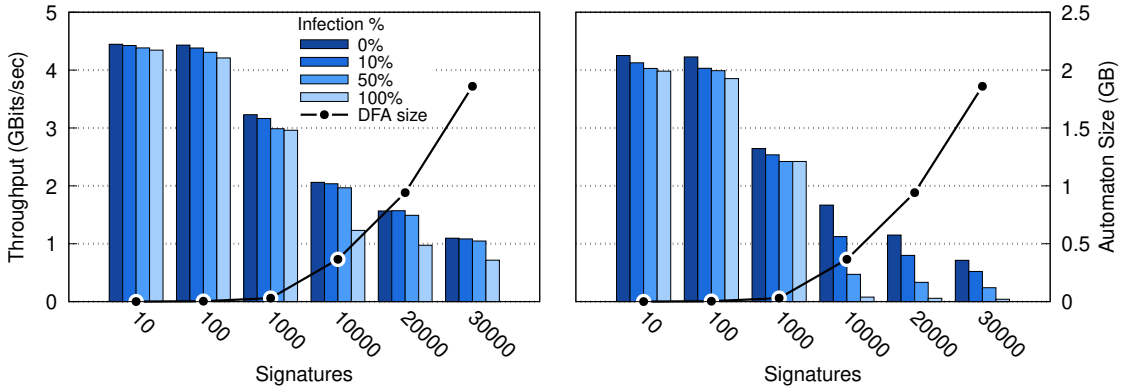
We evaluate the performance of our system using real malware signatures utilised by ClamAV. Specifically, we generate sets that contain a varying number of randomly selected signatures (i.e., 10, 100, 1000, 10000, 20000 and 30000), extracted by ClamAV's malware signature database. Each set is compiled into a separate automaton, as described in Section 4.4. In addition, we generate four different input streams for each signature set, containing 512MB of data. These streams are composed of various files, injected with signatures, so that they report 0%, 10%, 50% and 100% matches. The first two streams represent expected scenarios when the system is deployed while the latter are used to test the maximum capabilities of our system. To stress the limits of TrustAV, all input streams that contain matches are crafted so that every signature of each corresponding set is present in the data and every automaton state is accessed at least once.

The input data streams are synthetic but contain data that trigger all the malware signatures used. We choose to evaluate our system using synthetic input to be able to control the infection rate as well as provide input that triggers the worst possible access patterns in the automata. In this way we can stress the limits of our system and provide a thorough evaluation. In the future, we also plan to evaluate the end-to-end performance of our system using real input streams collected by various threat intelligence sharing platforms, such as STIX/TAXII [141].

4.5.2 Malware Analysis

Figure 4.3(a) illustrates the sustained throughput achieved by our malware analysis system when executed without using Intel SGX enclaves, while Figure 4.3(b) presents the performance, when TrustAV is entirely executed within secure enclaves. In addition, each automaton used in each execution is located inside the enclave memory space. Both figures display the throughput for various signature set sizes and different inputs, resulting to a varying number of matches.

Comparing the results, we can see that securing the execution within SGX enclaves adds no more than 19% overhead in execution time when the signature set is able to entirely fit in the EPC. This overhead occurs due to: (i) the execution of the added CPU



(a) Vanilla (b) SGX Enabled

Figure 4.3: Throughput evaluation of our malware scanning engine when executed with and without Intel SGX enclaves. Six different signature sets are tested against custom input streams infected by 0%, 10%, 50% and 100%. The Y2 axis indicates the DFA size.

instructions, (ii) the accesses inside the enclave's encrypted memory, and (iii) the time required to enter and exit an enclave. On the other hand, we can see that the overall throughput is reduced for automata exceeding the EPC size when the input streams are infected by 50% or more. In such cases, the multiple frequent random accesses to every automaton state produce a high number of accesses to EPC pages stored in untrusted memory. This behaviour triggers the costly process of restoring them in EPC and evicting others that might be shortly needed after processing only a few bytes from the input. Despite the fact that this issue is not observed with low input stream infection rates, it should be addressed as it severely affects highly infected input data streams by reducing the overall performance by an order of magnitude and could be possibly exploited for DoS attacks.

To identify the performance penalty introduced by protecting the signature automata, storing them inside the SGX enclave, we re-evaluate the throughput of our malware scanning engine with a different setup. In this case, we execute the TrustAV server with enclave support but we store the automata in unprotected memory in plain-text format. Since SGX enclaves have full access to unprotected memory with minimal overhead, our engine is still able to process the input streams without exposing the user's data which are still secured and processed inside the enclave. The results of this analysis are presented in Figure 4.4. Comparing the sustained throughput achieved with automata that exceed EPC size against the performance reported in Figure 4.3(b), we notice that with this setup the overall performance is significantly increased, exceeding 1Gbps for almost all cases. This benchmark also provides us with useful information for the implementation and evaluation of the caching systems, described in Section 4.4.3, as this is the maximum single-

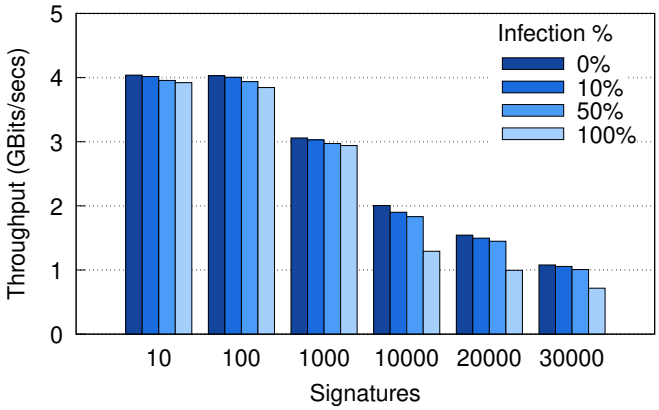


Figure 4.4: Performance evaluation of TrustAV when executed within enclaves while the automata are stored in untrusted memory in plain-text format.

threaded performance that could theoretically be achieved if the EPC page swapping is minimized by 100%. If the TrustAV hosting facility can be completely trusted and the integrity of the signature sets can be guaranteed or in cases where system performance is a strong requirement, this setup can still be utilized as user data are still never exposed out of the secure enclaves. However, it is not the optimal scenario as potential tampering of the signature set is possible and in such cases frequent integrity checking of the automata should be performed in parallel as a minimum countermeasure.

4.5.3 Performance Optimization Analysis

In this section, we evaluate the performance of the two caching systems that we implement to address the performance penalty introduced by severe EPC page swapping, discussed in Section 4.5.2. By analysing our system's performance with the automaton protected inside the secure enclaves as well as stored in untrusted memory in plain-text format, we are able to identify the introduced overhead for protecting the automaton and the possible maximum performance.

Aiming to preserve TrustAV's security and privacy guarantees, we explore the possibility of encrypting each automaton state using AES-GCM, provided by the Intel SGX SDK, and storing the automaton in unprotected memory in cipher-text format. In this setup, every referenced state is only decrypted inside the SGX enclave during processing and remains protected but without being allocated in EPC pages. We evaluate this setup for automata exceeding the EPC's capacity using input streams containing only infected files and present the results in Figure 4.5. While this setup eliminates the need for EPC page swapping, we notice that the performance is lower compared to storing the automaton in-

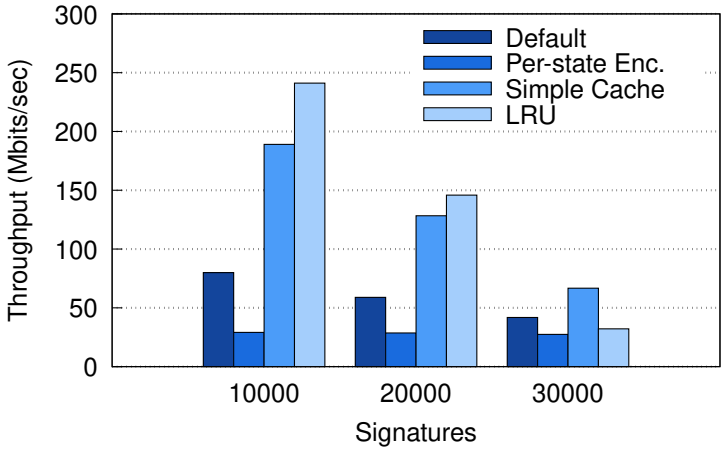


Figure 4.5: Performance comparison of the four ruleset protection schemes when the automaton size exceeds the ECP's capacity and the infection rate is 100%.

side the enclave and letting the SGX driver perform EPC page swapping. The main reason for this is that the decryption of each state using AES-GCM is slower than EPC page swapping, which is backed by the MEE implemented in hardware. Moreover, we notice that this is also the worst case performance for a cache implementation with minimum complexity, operating at 100% cache misses. After exploring the theoretical maximum and minimum performance boundaries of our system, we implement the two caching systems described in Section 4.4.3 and present their performance evaluation in Figure 4.5. For this evaluation we use the three signature automata that exceed the EPC memory limits, processing the input streams containing only 100% infected files that produce the worst memory access patterns in the automata. For the simple cache setup, we store up to 90MB of each automaton in the EPC, leaving enough memory for the application code, the batches of input data and the malware report generation process. The rest of the automaton is stored in cipher-text format in unprotected memory. Upon a cache miss, the state is fetched from the untrusted memory, decrypted inside the enclave and discarded after usage. We notice that this setup eliminates EPC page swapping while at the same time reduces the accesses to encrypted states in untrusted memory as it offers high cache hit rate, especially for the automata containing 10000 and 20000 patterns. More specifically, we notice that for the utilized rulesets, caching 25% of the automaton yields an 85% cache hit ratio, meaning that performance can be significantly improved by caching only a small portion of the automaton, as shown in Figure 4.5. However, this performance benefit gradually decreases as the automaton increases in size and a lower percentage of its states can be stored without violating EPC memory limits.

We conclude our evaluation by presenting the sustained throughput achieved by re-

placing the simple caching system with an LRU cache and executing TrustAV with the same automata and input streams as described above. As we can see in Figure 4.5, the LRU cache yields the best results for the automata containing 10000 and 20000 patterns by further increasing the cache hit ratio, efficiently utilising the available EPC space. However, we notice that it cannot outperform the simple cache when processing the input stream using the automaton containing 30000. The reason for this behaviour is that with the given EPC memory limit, the LRU cannot store enough states to provide substantial cache hit rate increase, compared to the simple cache, while LRU cache misses are more expensive since at each cache miss a state has to be evicted and a new one has to be stored, constantly updating the LRU data structures. In the future we aim to implement and evaluate more caching schemes (e.g., MRU) to address this, non critical, worst case scenario.

4.6 Summary

In this chapter, we presented our framework's first component, a practical and privacy-preserving cloud-based malware detection solution. TrustAV offloads the intensive process of malware analysis to a remote server with Intel SGX support to protect offloaded personal data as well as the malware scanning execution against untrusted parties. TrustAV is capable to perform with a minimum performance overhead, which is introduced by the utilization of hardware assisted enclaves. To reduce the memory footprint of our implementation, an important limiting factor for the majority of equivalent state-of-the-art solutions, we develop a caching scheme that eliminates EPC page swapping and offers up to 3x speedup, compared to the unoptimized implementation. With this component, we make the following contributions:

- We propose a practical cloud-based malware detection solution that provides users with strong privacy-preserving guarantees regarding the processing of their personal and sensitive data remotely by utilizing hardware assisted enclaves.
- We identify and present the performance constraints that are introduced by the Intel SGX technology, specifically in relevant signature-based analysis systems (such as intrusion detection or malware detection systems).
- We reduce the enclave memory footprint of our implementation, a limiting parameter for the majority of signature-based solutions in the state-of-the-art, by developing a caching scheme that eliminates EPC swapping and offers up to 3x speedup.
- Our proposed architecture enables the protection of signature-based automata that exceed the enclave memory limits in architectures where swapping of protected memory pages is not supported.

4.6.1 Discussion

Our proposed cloud-based malware scanning solution relies on a pattern matching technique to identify infected data. As discussed in this chapter, such approaches can face performance limitations when the automata required for the data processing are not able to entirely fit in secure enclaves and the input data are highly infected. Also, it is possible that, in certain applications, our component's server might operate with more than 30000 virus signatures in total when the underlying hardware only supports SGX version 1. Depending on the use case, preliminary testing should be applied to identify the properties of the automaton as different rules might generate automata with different caching characteristics while the access patterns on each automaton depend on the nature of the input data. As future work, we plan to extend our system so that the server can identify which rules should be applied on each input stream. In this way, the server will be able to utilize subsets of the ruleset so that smaller automata are generated, providing smaller memory footprints and better caching properties. We expect that this technique will further improve our system's overall performance, regardless of the infection rate of the input data.

Chapter 5
Kernel Integrity Monitoring

In this chapter, we present the second component of our security framework, named SGX-Mon, an external kernel integrity monitor that verifies the operating system's kernel integrity using a very small TCB without requiring any OS modifications or external hardware. SGX-Mon is a snapshot-based monitor, residing in the user space, and utilizes the trusted execution environment offered by Intel SGX to avoid detection from rootkits and prevent attackers from tampering its execution and operation-critical data. Our system is able to perform scanning, analysis, and verification of arbitrary kernel memory pages and memory regions and validate their integrity. The monitored locations can be specified by the user and can contain critical kernel code and data. SGX-Mon scans the system periodically and compares the contents of critical memory regions against their known benign values. Our experimental results show that SGX-Mon is able to achieve 100% accuracy while scanning up to 6,000 distinct kernel memory locations.

5.1 Design

Our kernel integrity monitor puts its integrity evaluation component, which represents its entire Trusted Computing Base, in an SGX enclave. Hence, the monitor is safe from attacks that can potentially compromise the Linux kernel and affect its execution. Moreover, attackers cannot inspect its code and identify that SGX-Mon exists in the system, scanning the kernel for malicious modifications. In addition, we use techniques to deprive an attacker from modifying the operating system's kernel to prevent running unauthorized code on the target system, as well as stop attacks that involve modifying the system's memory layout (e.g., through changing virtual memory mappings). This is an important step towards complete security protection of the kernel.

In its core, the integrity monitor is snapshot-based and provides programmability and easy deployment. During the secure bootstrap phase, SGX-Mon obtains the benign values of selected kernel memory regions that are not expected to be modified during normal execution. Then, it hashes these values and securely stores them inside the SGX enclave.

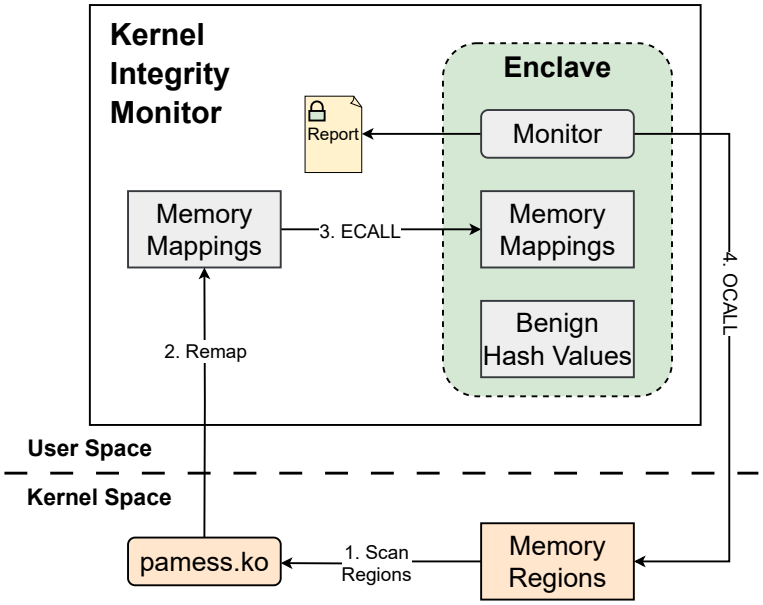


Figure 5.1: Architecture overview of the kernel integrity monitor.

During its execution, SGX-Mon periodically rescans these regions, computes their hash values and compares them with the benign ones. If a value is found to be modified, the system reports the existence of a possibly malicious action. Besides the static text parts of the kernel or the already loaded LKMs that can be easily hashed, the OS kernel consists of additional parts that frequently change; for example the VFS layer's data structures change when new file systems are mounted or removed. Also, every LKM can add function pointers. The memory regions to be monitored can be specified by the user and can include pages that contain kernel text, loadable kernel modules, or function pointer arrays (i.e., jump tables). Our system's architecture is depicted in Figure 5.1.

5.2 Implementation

5.2.1 Mapping Kernel Memory to SGX Enclaves

During its secure bootstrap phase, the integrity monitor needs to acquire the kernel memory regions that need to be monitored. Since these regions are located in the kernel virtual address space, the first step is to map them to the address space of the user process that issues the execution of the secure enclave which performs the monitoring.

To provide the desired integrity monitoring functionality from the user space, we need to develop a mechanism to reference kernel memory regions. The Linux kernel prohibits user space applications to directly access memory regions that have not been assigned to them. Typically, all memory accesses to pages that are not mapped to the process's

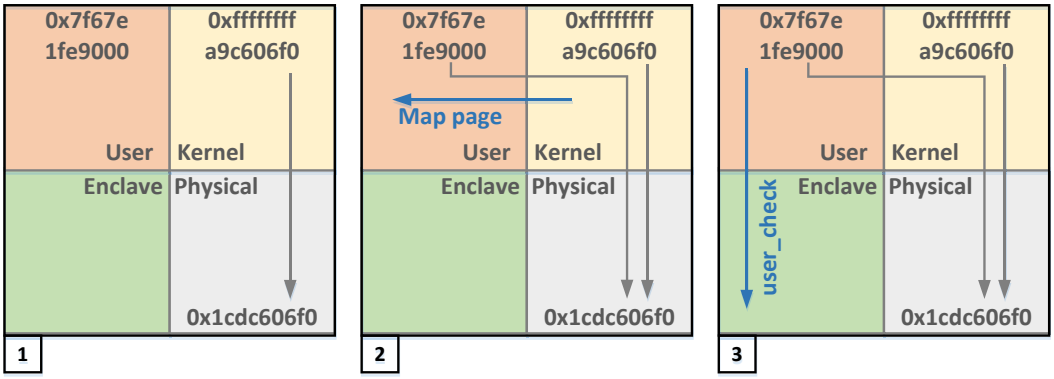


Figure 5.2: Mapping OS kernel memory to the address space of the integrity monitor. In *step 1*, we locate a desired kernel virtual address pointing to a physical address. In *step 2*, we duplicate this mapping to user space using our page table manipulation kernel module. In *step 3*, we pass the user space virtual address to the SGX-enclave using the `user_check` option.

virtual address space result in a segmentation violation since they are considered illegal. For this reason the memory regions where the kernel and its data structures reside have to be mapped to the integrity monitor's address space.

To bypass this OS functionality without modifying the operating system kernel, we develop a loadable kernel module, named `pamess`, able to map user-specified memory regions to user space. These regions can correspond to pages that contain kernel function pointers, data, as well as LKM or kernel text. The virtual addresses of the aforementioned memory locations are obtained via the `/proc/kallsyms` interface. In our case, the Linux kernel under test is built with `CONFIG_KALLSYMS=y` and `CONFIG_KALLSYMS_ALL=y`, so no recompilation of the kernel is required. However, these two options are not a strict requirement for the proper operation of our system.

Since the kernel symbol lookup table is present, we are able to simplify the development in two ways. First, no custom memory scanner is required to locate all the kernel memory regions subject to monitoring. Second, it allows us to easily locate the address of the kernel page table by acquiring the address of the `init_mm` symbol without explicitly exporting it via modifying the kernel. In this case, no modifications to the host's operating system are required and our system is able to operate at its full potential just by loading our custom kernel module. In scenarios where the access to the kernel lookup table has to be restricted, we would locate the various memory locations using either an external symbol table or via a custom memory pattern scanner. The `pamess` kernel module is only required during the secure-bootstrap phase, to acquire the correct memory mappings, and can be unloaded afterwards as it is not needed during SGX-Mon's execution.

The process of making the aforementioned memory pages available to the SGX enclave is depicted in Figure 5.2. During the secure bootstrap phase, we allocate a data structure inside the SGX enclave responsible for storing all the available data and metadata of each kernel region, such as the correct checksum, number of required pages, offsets, etc. Our kernel loadable module resolves the physical mapping of each desired kernel virtual address and instructs the integrity monitor to allocate the appropriate amount of pages in its address space, as shown in *step 1*. Then, in *step 2*, the module makes the allocated page to point to the same page as the kernel virtual address by duplicating the PTE in the user-page table. Finally, in *step 3*, we update the data structure inside the enclave by passing a pointer to the virtual page of the monitor's address space using the `user_check` option. By issuing read operations of the appropriate size, the enclave side of the monitor is now able to inspect the contents of the desired kernel memory regions.

However, an attacker could manipulate the kernel memory mappings, defined by page tables, to fool the SGX-enabled integrity monitor and bypass detection. The attacker could manipulate the `CR3` register and map the kernel code to a different physical page. In this work, we assume that the kernel runs in a known state and is not compromised prior to the secure bootstrap of the monitor, thus by monitoring the `CR3` register we limit the possibility of such an attack.

5.2.2 Kernel Integrity Monitoring in SGX

The process of monitoring the specified memory locations is entirely performed inside the SGX enclave. The first part of the process is acquiring the correct checksums of the specified memory regions. During the secure bootstrap phase, once the page mapping process is finished, the monitor obtains the hashes of the contents of all specified locations and stores them in its data structures. Since this operation is performed during a secure bootstrap phase, the state of the entire system is considered benign, thus these checksums are considered a reference point for the system's security.

After the process of obtaining the checksums is finished, we assume that the secure bootstrap phase has finished and the system can be put at risk at any time. The monitor operates as a daemon, running in an infinite loop, constantly iterating through all the memory regions and hashing their contents. The hashes are compared for changes against the values obtained during the previous phase. Since the entire checksumming process, as well as the clean-state checksums, reside inside the SGX enclave, the monitoring operation remains untampered.

SGX-Mon is able to perform the checksumming operations by using either the CRC-32 or the SHA-256 hashing algorithm. In its default operation, the monitor opts for CRC-32, implemented in accordance with ISO 3309. We choose CRC-32 mainly due to its speed, simplicity, and ability to cover a large number of individual memory locations without a

severe performance overhead that penalises the safety of the system (see Section 5.3 for performance comparison). The system is also able to perform the monitoring by entirely using SHA-256 in scenarios where CRC-32 is not considered sufficient enough to provide collision free results. However, SHA-256 is significantly slower and while it offers higher collision resistance it is not able to cover the same amount of kernel memory regions in the same frequency. In such cases, the integrity monitor can be tuned to use CRC-32 for the majority of memory locations and SHA-256 for regions that are regarded to be highly security critical. However, SGX-Mon can be easily extended to provide a wider range of hashing algorithms.

5.3 Evaluation

In this section we present the evaluation of our SGX-enabled Kernel Integrity Monitor in terms of performance and accuracy. We explore the characteristics of CRC-32 and SHA-256 in respect to the detection rate of a kernel-side attack. We choose to evaluate SGX-Mon with CRC-32 and SHA-256 as the former stands as a good representation of the fast hash generation algorithm family while the latter provides higher collision resistance with the penalty of lower performance. Other algorithms, such as MD5 or SHA1, yield performance and collision resistance results between these bounds. Moreover, we measure the impact of each algorithm to the system's overall performance and ability to identify such attacks by covering a reasonable amount of kernel memory regions.

5.3.1 Evaluation Setup

The system used for the evaluation of our kernel integrity monitor consists of an SGX-enabled Intel Core i7-6700 CPU, clocked at 3.40GHz, and 16GB of DDR4 memory, clocked at 2400 MHz. The system is running Arch Linux with kernel version 4.14 and the latest version of the SGX1 software stack.

5.3.2 Snapshot Frequency

We begin the evaluation of our kernel integrity monitor by identifying the achievable snapshot frequency of each hashing algorithm in respect to the number of kernel memory regions provided. In this set of experiments, the system is instructed to monitor 8-byte long kernel memory regions, obtained using the `/proc/kallsyms` interface. Setting the system to its default mode of operation, using CRC-32, we perform constant monitoring of the provided memory regions while measuring the sustained snapshot frequency. We conduct the same experiment, each time increasing the number of monitoring addresses by 2,000, starting with 1,000 entries up to 20,000. The outcome of this evaluation is depicted in Figure 5.3. As expected, we notice that as the number of memory locations in-

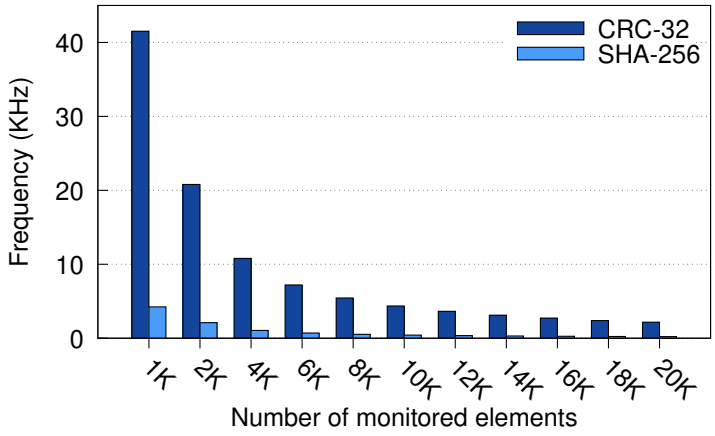


Figure 5.3: Sustainable scanning frequency using CRC-32 and SHA-256 in respect to the number of monitored regions.

creases, the frequency of monitoring the locations decreases. Overall, the system is able to achieve a frequency of 42 KHz when monitoring 1,000 distinct kernel memory regions and provides monitoring frequencies greater than 10 KHz for configurations containing up to 4,000 memory locations.

We also perform the same set of experiments, this time by tuning the integrity monitor into using SHA-256 for the checksumming operations. The results of this experiment are also shown in Figure 5.3. We observe that SHA-256, being a more computationally intensive algorithm compared to CRC-32, yields lower sustained monitoring frequencies. On average, its performance is one order of magnitude lower than CRC-32 and SGX-Mon achieves a scanning frequency of 4.3 KHz when monitoring 1,000 kernel memory regions.

5.3.3 Monitoring Accuracy

After obtaining the performance characteristics of our system using the two available hashing algorithms in multiple configurations, we want to determine its ability to accurately identify kernel-side attacks. We base this analysis on the monitor's ability to detect the presence of a malicious self-hiding kernel loadable module. Similar case studies have also been conducted for the evaluation of other kernel integrity monitoring systems [112, 121].

To perform this evaluation, we develop a custom self-hiding loadable kernel module (LKM) that performs zero malicious operations other than deleting itself from the loadable kernel modules list after it is being loaded. With this setup we are able to stress SGX-Mon to the maximum extend since this artificial module does not execute any code useful to an attacker, thus being exposed to the system for the minimum possible extend.

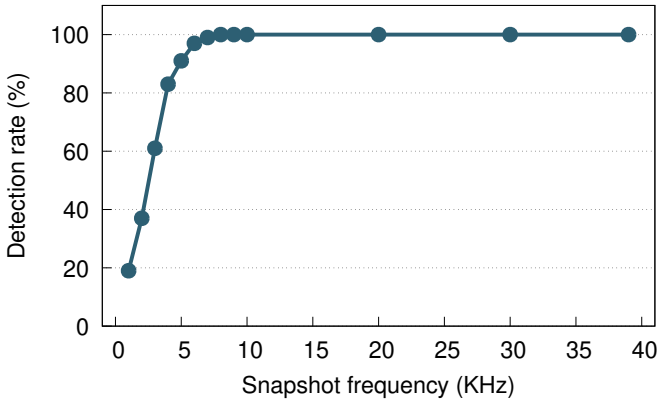


Figure 5.4: Detection rate of the self-hiding LKM with different snapshot frequencies. Frequency configurations of 8 KHz or more achieve a 100% detection rate. For each frequency configuration we monitor the head of the kernel modules list and load a module that deletes its entry from the list 100 times.

In typical Linux environments, loadable kernel modules are handled using the various available utilities, such as `insmod` and `rmmod`. These utilities are responsible for loading the modules into memory and invoking the appropriate system calls for their initialization. The reverse operation is performed when a module needs to be unloaded from the system. During the initialization phase, the `insmod` tool opens the module object file and invokes the `finit_module` system call. This operation adds a handle to the kernel's loadable module list data structure, initializes any parameters given to the module and invokes the module's `init()` function. All modules currently present to the system can be obtained by iterating through this list. Once our artificial module is loaded, it hides itself from the system by removing its entry from the kernel's loadable module list. This transient malicious operation can be detected by our kernel integrity monitor by scanning the head of the loadable kernel modules list with a sufficient frequency.

With the following experiment we determine the minimum required monitoring frequency for the successful detection of such modifications of the kernel's data structure. We measure the detection rate using multiple frequency configurations while loading the self-hiding module 100 times. The outcome of this experiment is displayed in Figure 5.4. We observe that SGX-Mon is able to reliably detect the presence of the malicious module with 100% accuracy when operating with a snapshot frequency of 8 KHz or more.

We conclude our evaluation by performing the first set of experiments, this time including the head of the kernel's loadable module list in each kernel memory region configuration, while at the same time loading the self-hiding module into the system. Figure 5.5 presents the comparison of the sustained detection rate when monitoring with CRC-32

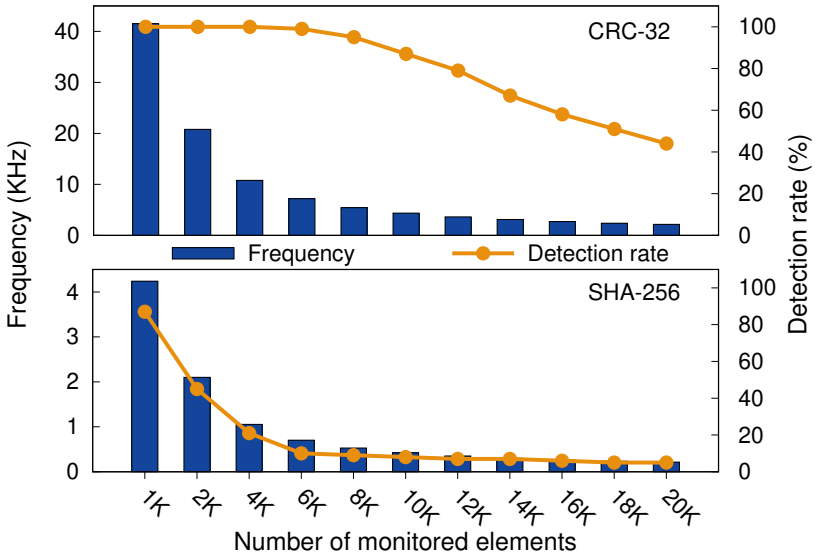


Figure 5.5: Scanning frequency and detection rate of the self-hiding LKM using CRC-32 and SHA-256 in respect to the number of monitored regions.

and SHA-256, using an increasing number of kernel memory locations. As we can see in the Y2-axis, our system is able to achieve 100% detection rate, using a single thread, while scanning up to 6,000 distinct memory locations by using CRC-32 for the checksumming operations. On the other hand, when SHA-256 is used, the system achieves a maximum of 87% detection rate while scanning 1,000 memory locations. The increased level of security provided by SHA-256 comes with a substantial penalty to the system's overall performance. Nonetheless, for our specific hardware, SHA-256 can be used to successfully monitor up to 850 distinct memory locations of significant security importance.

5.4 Summary

In this chapter, we presented our framework's second component, an external snapshot-based integrity monitor able to identify kernel-side attacks that tamper and modify critical Linux kernel memory regions and data structures. Our system is executed in the user space, without any need for external hardware, kernel modifications or hypervisors, and resides inside a secure Intel SGX enclave. The module is able to remain untampered and undetected by malicious third parties and operates with a minimal TCB. SGX-Mon is able to achieve up to 100% accuracy while scanning up to 6,000 distinct memory locations using CRC-32 and up to 87% accuracy while scanning up to 1,000 memory locations using SHA-256. With this component, we make the following contributions:

- This module is the first, to our knowledge, kernel integrity monitor that leverages Intel SGX to protect its code and data from identification and modification. SGX-Mon utilizes a very small TCB, able to be contained in a single SGX enclave and is easily audited.
- SGX-Mon is purely software-based, running in the user space, and does not require external or non-commodity coprocessors, devices, TPMs or hypervisors. This restricts potential memory leaks and vulnerabilities that are introduced by extra hardware or complex system management software.
- We evaluate the effectiveness of our system in identifying transient kernel-side attacks and study the appropriate monitoring intervals that guarantee that transient rootkits are not able to perform any malicious actions and still be able to remain undetected.

In the future, we aim to explore the effectiveness of random intervals to increase the system's overall accuracy when scanning more memory locations using highly collision resistant algorithms, such as SHA-256. Moreover, we plan to extend our system with a rule engine able to perform specific user-defined actions when a threat is detected as well as identify complex attacks targeting kernel code and registers. Also, we plan to identify our system's maximum effectiveness and performance to CPU utilization ratio using a multi-threaded setup.

Chapter 6
Enhancing a Modern Operating System

In this chapter, we present Andromeda, our security framework's component that provides secure enclaves for the Android operating system. Developers can explicitly use them for their applications, either via the native API in C/C++ or our Java interface that provides access to the secure enclaves through JNI bindings. In contrast to other TEE approaches targeting the Android OS, Andromeda has the potential for multiple enclaves in a system simultaneously, making it more flexible for general-purpose security-critical operations, offering per-application or per-function isolated secure environments.

In addition, Andromeda implements popular Android services, enhanced with enclave capabilities, hence securing and protecting their functionalities. We provide two representative services: (i) a secure key management system, and (ii) a data protection scheme for data flows, that enhance the security of Android OS and offer protection schemes for several applications that deal with sensitive data (such as cipher keys, personal data, medical data, etc.). These services enable Andromeda to support an efficient and robust end-to-end encrypted data flow model in which external devices that pair with Android can securely transfer and process their data on the Android device, or even pair with a remote Android cloud-based service.

We have currently implemented the Andromeda prototype for Intel processors with SGX1 support. Any device that is equipped with an SGX-enabled processor can run Andromeda natively, out of the box, including handheld devices, convertibles, set-top boxes, and car entertainment units. However, we have to point out that Andromeda is not bound to Intel SGX; instead the proposed mechanisms could be implemented on top of other architectures offering secure user-level enclaves. For instance, there are approaches that implement user-level secure enclaves, compatible to SGX, either independent of the underlying CPU (such as Komodo [83]) or on top of ARM TrustZone (such as Sanctuary [55]). Andromeda is not fundamentally tight to Intel SGX and, as such, could be implemented on top of such approaches instead. Besides that, we note that a number of vendors are developing similar hardware protection mechanisms, including Apple Secure Enclave [38] and IBM SecureBlue++ [53]. Even though these mechanisms are not identical, many of

the proposed techniques of Andromeda can be adapted to use these hardware features, the need of which will increase in the future.

6.1 The Android OS

Android is an operating system mainly designed for small handheld smart devices, including but not limited to mobile phones, tablets and watches. It is being developed by Google LLC, first released in 2007, and is currently the most widespread OS for smart devices [15, 103]. Android's backbone is based on the Linux kernel, thus granting it extensively tested security features and stability, and also allowing developers and manufacturers alike to develop hardware drivers for a well known kernel. Google also had to make a few additions to provide a more customised kernel functionality for Android's requirements. A few key additions are the `wakelocks`, a power management component crucial for mobile devices, a unique out of memory (OOM) handling, also informally known as *Viking Killer*, the `ashmem`, a new shared memory allocator for low-memory devices, `pmem`, a process memory allocator and also `Binder`, an Android specific interprocess communication mechanism and remote method invocation system, essential to Android due to the fact that it does not support the use of the Linux SysV IPC [26].

Android is built on top of the Linux kernel with components such as the hardware abstraction layer (HAL), which provides various standard interfaces that allow higher Java APIs and code to make use of a device's hardware components, and the Android Runtime (ART), a special virtual machine similar to Java's JVM, designed to run on low-memory devices. There are also native C/C++ libraries and both HAL and ART are written in C/C++. However these native libraries do not provide the same functionality as they would in a traditional Linux machine. On the top layer of the Android architecture, there is the Java API Framework, which provides applications a means to access the other layers in a constant way throughout different machines. All Android applications, while able to use native C/C++ code, are developed in Java, enabling them to be executed on multiple and different devices. The majority of cryptographic operations in Android, including encryption, decryption, message authentication (MAC), key generation and agreement, are handled by the Android Keystore [74], that also provides a central place for storing cryptographic keys for all applications. Keymaster is a part of the Android Keystore service responsible for generating new keys for encrypting, decrypting and hashing data. It supports various cryptographic functions like AES, RSA, SHA and more. To generate such an encrypted key for an application and perform cryptographic operations, one has to generate a `SecretKey`, initialize a `Cipher` with the desired mode (encrypt, decrypt or other), and choose the appropriate algorithm and its properties for the current operation. Android defines an abstract programming interface that can be used for the third-party implementations, plugged in seamlessly as needed. Therefore application developers may

take advantage of any number of provider-based implementations without having to add or rewrite code.

6.2 Threat Model

In this work, we assume a powerful and active adversary who has root privileges and access to the physical hardware. The adversary can control the entire software stack, including the OS kernel and other system software. However, we explicitly exclude denial-of-service (DoS) attacks on enclaves, given that the design of SGX allows the host OS to control an enclave's life cycle anyway. As a result, an attacker can prevent or abort the execution of enclaves, but should not gain any knowledge by doing so. Moreover, side-channel attacks that exploit timing or page faults or attacks based on vulnerabilities of the application running inside the enclave are proven to be feasible on SGX enclaves. However, protecting SGX enclaves from side-channel attacks that either focus on software or hardware bugs is orthogonal to Andromeda and thus we consider them as out of scope of this work. However, any successful attempt to protect SGX-enabled software/hardware has a direct benefit to our system. Finally, we assume the design and implementation of SGX itself is secure and does not contain any software or hardware vulnerabilities.

6.3 Design

Our objective is to offer secure enclaves for the Android OS to protect sensitive services from the threats defined in Section 6.2. This enables Android developers to explicitly leverage them for their applications. We also want to utilize secure enclaves inside Android services that operate on sensitive data (such as Keystore), so they can be used transparently by applications with forward and backward compatibility. Overall, Android developers should be able to build their applications and use the secure enclaves as transparently as possible, ideally without writing extra code or heavily modifying existing applications.

An enclave cannot be initiated on its own and the Intel Launch enclave must be used to generate the appropriate launch token. In addition, an enclave's code always has to be executed in Ring-3 with a reduced set of allowed instructions and a limited amount of available memory. Thereby, we decide to build an architecture that runs solely in user space, providing the interface and the services that Android applications can use in an expressive and flexible way. Figure 6.1 gives an overview of the Andromeda architecture. It comprises of different layers that can be used by different kinds of applications for different purposes. Using these mechanisms, we enhance popular Android services, such as the Device Pairing and Keystore service, to leverage secure enclaves internally to increase their security in a robust and transparent way. Finally, we also implement an environment, within SGX, so external devices that have paired with Android can securely transfer and

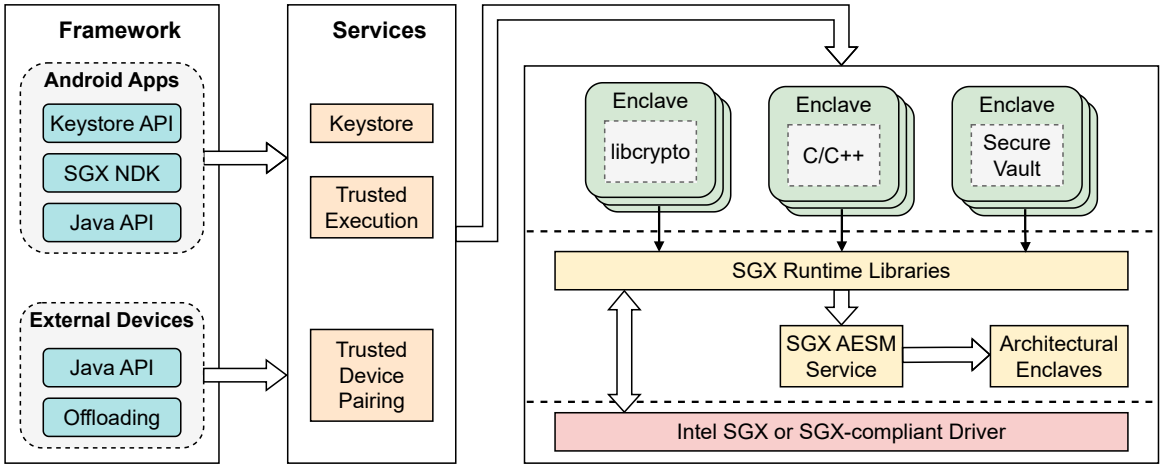


Figure 6.1: Andromeda architecture overview.

store sensitive data on the Android device. Andromeda is responsible to protect all sensitive data by encrypting them across the full path from the external device to the Android OS. Further, Andromeda optionally enables the processing of these data via functions that the data-publishing applications have submitted for execution in the SGX enclaves.

6.3.1 Trusted Execution and Storage

Andromeda provides trusted execution and data storage services on top of SGX. The services can be used by local Android apps, as well as from remotely paired devices, as described in Section 6.3.2. At the lowest level, applications can use the native API provided by the SGX runtime libraries, to achieve the maximum performance. The process of utilising secure enclaves in an application developed in native C/C++ code remains the same as for every other native C/C++ Android application. The developer needs to prepare and integrate the Intel SGX counterpart of the application (similar to the Linux environment) and then cross-compile the application with our custom Android toolchain, which is able to handle the compilation of both trusted and untrusted parts of the code. Developing Intel SGX enclaves for an APK implemented using Java requires the use of JNI bindings. For this reason, we provide a Java API (described in Section 6.5.3), which wraps the SGX functionalities in appropriate classes. The developer needs to extend these classes with methods that will be eventually executed in the application's enclave and compile the code using the Andromeda toolchain, which also provides JNI bindings for each SGX-enabled function requested. In this way, the developers can easily interface with the enclaves from the APK level. Moreover, Andromeda provides a secure data vault system and exposes a simple Java API for Android applications. Using the data vault service, applications can securely store data inside the SGX enclaves or seal them for secure file system storage.

6.3.2 Andromeda Services

Keystore Service

The main purpose of Android Keystore is to store cryptographic keys and offer cryptographic operations in a secure container, protecting them from tampering. However, if not implemented with secure hardware support, it is vulnerable to a broad set of attacks, as described in Section 6.2. Having the secret and private keys stored in clear-text makes them an easy target for malicious software running on the device. Andromeda offers the mechanisms to keep the secret keys in a protected space, within secure enclaves, thus solving and overcoming leakage scenarios and cold boot attacks.

The Keystore is implemented in C/C++ while Android uses a binder to communicate with the Java counterpart. Internally, the Keystore can handle various types of entries. Some of them are `PrivateKey`, `SecretKeyEntry`, and `TrustedCertificateEntry`. Each entry is identified by an alias name which corresponds to the Keystore entry. When generating such an entry, it is possible to choose from a range of available cryptographic algorithms or use the default. In this way, the Android Keystore is able to store multiple keys simultaneously, regardless of type, name and algorithm. At the same time, different applications can utilize the Keystore and store their keys without having to deal with collisions.

An overview of our SGX-enabled Keystore operation is illustrated in Figure 6.2. A major advantage of Andromeda Keystore is that it can be used even by legacy apps without any code modifications or recompilation. The simplest way is to have the entire Keystore inside a single enclave. However, this design leads to a large TCB that is generally harder to review, or possibly verify, and is assumed to have more vulnerabilities. To overcome this problem, we place only three core operations in the secure enclaves, which are used by the majority of cryptographic algorithms: (i) the key generation, (ii) the data encryption, and (iii) the data decryption. By doing so, we ensure that all private and secret keys reside in secure enclaves while having a small TCB that can be easily verified. The memory for the keys is allocated inside the SGX enclave and only their pointers are returned to the user space, preventing any attempt to read them, extract them or modify them, even via physical access to the device's DRAM.

Our current implementation uses RSA-1024 and AES-CTR; we note though that other modes can be easily implemented. AES divides each plain-text into 128-bit fixed blocks and encrypts each block into cipher-text with a 128-bit key. The encryption algorithm consists of 10 transformation rounds. Each round uses a different round key generated from the original key using Rijndael's key schedule. The whole encryption and decryption occurs inside the SGX enclave, ensuring that keys and all intermediate states are well protected. Similarly, we have implemented RSA encryption and decryption.

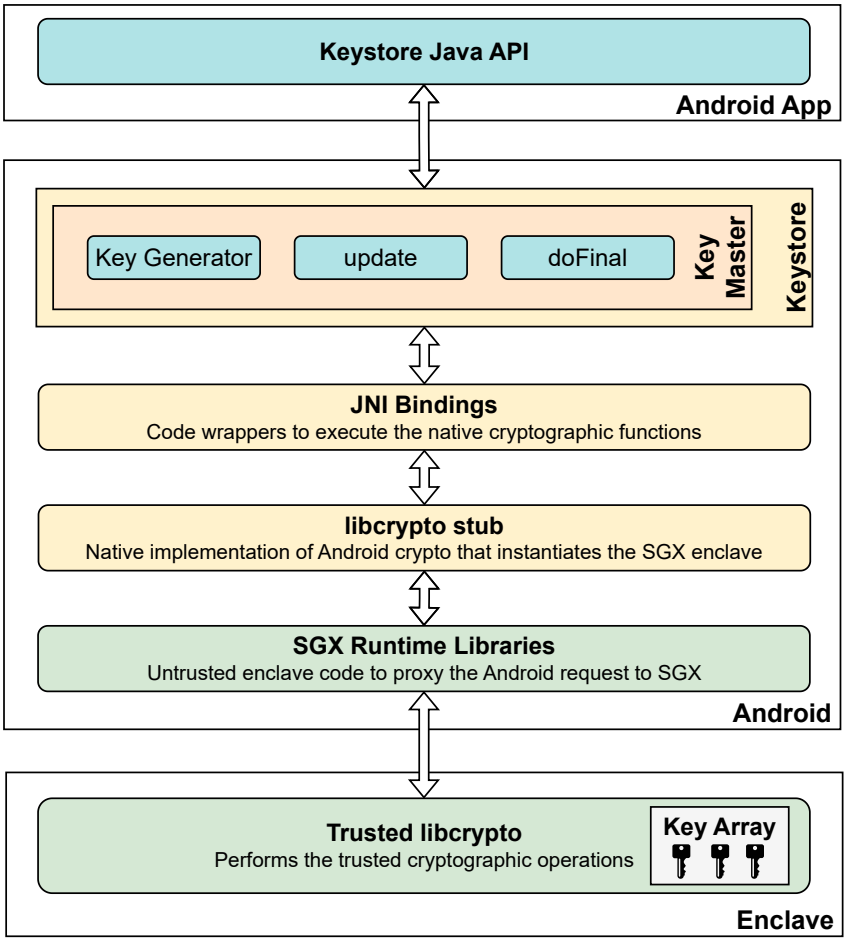


Figure 6.2: The Andromeda Keystore architecture. Cipher keys are stored only in SGX enclaves. Encryption and decryption is performed using the default Keystore API, internally redirecting to Andromeda's trusted implementation.

Trusted Device Pairing

Andromeda provides secure device pairing between devices, even when only one (i.e., the Android device) is equipped with an SGX-enabled processor. Such scenarios are typical when small external devices, such as sensors and wearables with limited security capabilities, need to be paired with more powerful Android devices (i.e., a smartphone or gateway). To accomplish secure device pairing and attestation in such use cases, Andromeda offers the functionality that enables the external devices to securely connect with the SGX-capable Android device. The main concept is that data-publishing wearable or external devices can protect their sensitive data, so they only reside and being processed within designated functions that run in SGX-provided enclaves.

First, Andromeda generates a key pair and distributes the public key to the external device and the corresponding private key to a local secure enclave. Each external device is assigned to its own secure enclave to ensure isolation between each other. These keys can be used later to establish a session key using Diffie-Hellman. The process of establishing and storing the keys is performed entirely inside SGX enclaves in the case of the Android device. We assume that the external device runs on a minimal codebase with limited I/O, thus the integrity of the key management can be attested and preserved. While this end-to-end encryption of the I/O channel ensures data protection during transfers, the need of attestation between the two devices remains a critical point to prevent malicious users impersonating as one of the two devices. In cases where the external device is capable to execute the Intel Remote Attestation process, it is able to verify that it is indeed communicating with a secure enclave, running on SGX-capable hardware without emulation. However, in some cases, Intel Remote Attestation cannot be performed due to the limited computing capabilities of many external devices. To overcome this, we utilize one-time passwords (OTP) instead, which are an essential part for our remote attestation alternative procedure. More specifically, we use Google key generator to create an arbitrary key that we can then register with a secure SGX enclave. The registration is performed at the first connection and Andromeda (optionally) prompts the user to verify the registration. Once the key has been successfully registered, the attestation procedure starts with the external device demanding a 6-digit OTP to be exchanged. The generated OTPs are based on RFC 6238 [22]. Upon receiving the OTP, the external device calculates an OTP with the same key. If both match, the external device can be certain that it communicates with the SGX enclave, since the entire OTP process is performed inside the enclave. Once the OTP is verified, the secure communication channel is established as described above.

6.4 Implementation

6.4.1 Setting up SGX for Android

Cross-compiling Intel SGX for the Android OS is an extensive and challenging task. Due to the complexity of the software and the many differences between a standard Linux distribution and Android, we have to split the porting process into several smaller tasks to constantly keep proving the potential and validity of our goal. For this reason, we perform the Android port in the following steps. First, we compile the SGX SDK for a different Linux distribution than Ubuntu, which is the officially supported distribution for Intel SGX, namely Arch Linux. Since Android is also based on Linux, this process lets us understand how different compiler and library versions affect the possibility of porting SGX on Android. Second, we validate that we can build the Android Open Source Project (AOSP) form scratch and successfully install and run it on an SGX-capable x86 machine. Finally,

we integrate the SGX functionality into the AOSP source tree by cross-compiling it and providing the necessary libraries for its correct operation.

The whole process of building the SGX environment for an officially unsupported Linux distribution, such as Arch Linux, is a quite tedious procedure. The main reason for this are the kernel, compiler, and library version incompatibilities since Arch Linux uses a rolling release system providing the latest version of each package, oftentimes a few major versions higher than what the SGX runtime expects. While analyzing the dependencies of the SGX SDK, we find the following to be essential for a standard enclave execution: (i) the SGX kernel driver, (ii) the `aesm_service`, which is a background daemon serving as a management agent for SGX enabled applications, (iii) the `libsgx_urts.so` and `libsgx_uae_service.so`, needed for executing enclaves in hardware mode, (iv) the `libsgx_urts_sim.so` and `libsgx_uae_service_sim.so`, needed for the software emulation mode, and finally (v) the `le_prod_css.bin` and (vi) `libsgx_le.signed.so`. By analysing the basic components of the SGX environment and their software dependencies, we manage to understand, in practice, the software requirements and the process of building it for an unsupported platform.

Porting SGX on Android is an even more complicated process. First, AOSP has to be built from scratch and installed natively on an SGX-enabled x86 machine. Then, porting the SGX environment requires a lot of effort since each change to the source tree requires the following: (i) rebuilding the Android image, (ii) flashing it on the host machine, and (iii) verifying the correctness of each change as well as the stability of the system. During the development of this system, utilizing virtual machines was not an option since no virtualisation system offered guest support for Intel SGX. After successfully booting Android on x86, we prepare the SGX environment to be ported.

Intel SGX1 requires to be built on a desktop-based Linux distribution that utilizes `GCC-5` and above while Google's NDK (Native Development Kit for Android) offers `GCC-4` and `clang`. The errors were excessive and most of the time could not be resolved so we needed an alternative compiler, closer to Intel's SGX requirements. To build Intel SGX for Android, we use the CrystaX NDK (Native Development Kit) [7], a drop-in replacement for Google's Android NDK, offering `GCC-5.3` compatibility.

The first task that needs to be performed is to configure the environment and the cross-compiler by exporting all the required libraries for the building environment. Intel SGX needs several libraries, such as `protobuf`, `libssp`, `curl`, and `ssl`, which we manually cross-compile from scratch. We also edit the configuration files of `gperftools` and `libuwind`, and manually set the `cross_compiling` field to *true*, to fix the incompatibilities between Android and Ubuntu header sources (which Intel SGX is targeted to). In addition, we remove all links to `libpthread`, since this library is integrated in the Android source. Moreover, due to the stripped-down kernel version that is used by the Android OS, the `RDRAND` instruction used by `sgx_read_rand` to perform random number generation is not available.

To overcome this issue we use a software-based implementation for random number generation that is fully compatible with the existing API and works both on Android and SGX.

After successfully cross-compiling the SGX source tree, the final step is to cross-compile the SGX kernel driver and port it to Android. Unfortunately, there are inconsistencies between the supported kernel used by Ubuntu and the Android kernel headers, thus, the signatures of several kernel functions are different. For this reason, some patches are required in order to build the driver while it also requires to be built in-source with Android. For this reason, we modify the Intel SGX driver code and add it in the Android kernel source tree, edit the corresponding KConfigs and Makefiles in the kernel source, and create a custom kernel configuration that includes our modified driver. By doing so, we manage to successfully build, install and run an Intel SGX-enabled Android x86 instance on bare metal hosts.

After completing all the steps described above, the final step is building a testing application that utilizes SGX enclaves in both hardware and simulation modes. Every enclave binary must be signed by the `sgx_sign` tool which is provided by the SDK, responsible for signing every enclave during compilation time. However, cross-compiling the whole Intel SGX SDK and PSW, produces a `sgx_sign` binary that is only executable on Android; preventing developers to build applications and sign them on their development environment. To address this issue, we rebuild the SGX source tree once more, this time using only Ubuntu's default tools, keeping only the `sgx_sign` tool which is now used by our cross-compiler toolchain. With this configuration, every SGX-enabled Android binary can now be cross-compiled and signed on the development host.

6.4.2 Running an SGX Application

An SGX application can run either in `hardware` or `simulation` mode. To make use of the underlying hardware and leverage Intel SGX as a service, we cross-compile SGX applications using `SGX_MODE=HW` which links against `libsgx_urts.so`. Of course, since these libraries are not available in Android's source tree they must be provided to the `LD_LIBRARY_PATH` of the corresponding application by exporting the paths of each one of them. Apart from the required SGX dependencies, the libraries that were linked during the SDK compilation must be also provided and exported to the `LD_LIBRARY_PATH` of the given application. Additionally, we use `insmod` to load the driver and then start the `aesm_service` on boot. The Android service system has several differences compared to Linux. Editing a system service file, like `init.d`, is not enough for Android to deploy a new system service. Instead, a new application, marked as a service, has to be created and meet specific code requirements [2]; i.e., all native functions of `aesm_service` need to be wrapped with JNI calls for it to be accessible by the Java part.

To overcome this issue, we simply adjust the `aesm_service` source code to run as a

daemon in the background and interact directly with the native part. The other solution would be to discard the whole Android application part and interact with the native part directly. By examining the source code of `aesm_service` we manage to run the application as daemon (which is essentially a service) so the app can start and stay alive. Whereas, if we start it without the specified input, it would just terminate with no output. Also, the `aesm_service` requires the `le_prod_css.bin` and `libsgx_le.signed.so` binaries to properly execute so we transfer these binaries from the Intel SGX output directory to the `aesm_service` directory in Android, before its execution. Finally, running a C/C++ application in Android requires it to be built with the `-pie` and `-fPIE` flags. These flags instruct the linker that the program's code can be executed regardless its absolute address. After all the aforementioned requirements are met, we are able to cross-compile and execute SGX-enabled Android applications.

Enclaves can be created using the `ECREATE` instruction, which initializes an SGX enclave control structure (SECS) in the EPC. The `EADD` instruction adds pages to the enclave, which are further tracked and protected by the SGX (i.e., the virtual address and its permissions). The `EINIT` instruction creates a cryptographic measurement, after loading all enclave pages. The cryptographic measurement can be used by remote parties for attestation. After the enclave has been initialized, enclave code can be executed through the `EENTER` instruction, which switches the CPU to enclave mode and jumps to a predefined enclave offset. The `EEXIT` instruction causes execution to leave the enclave.

6.5 Andromeda Framework

The Andromeda framework is split into three parts: (i) the enclave-enhanced Android Keystore, which can be utilized transparently, (ii) the native C/C++ API, used to initialize and configure SGX using native C/C++ code, and (iii) the Java API, which provides a set of building blocks for APKs.

6.5.1 Andromeda Keystore

Android apps can transparently utilize the Andromeda Keystore service to securely perform cryptographic operations. Private keys and other sensitive information are kept in an encrypted format in an array that resides in SGX memory and cannot be accessed in any way by the host. To perform a cryptographic operation: (i) the required (encrypted) key is fetched from the array, (ii) it is decrypted inside the enclave, and (iii) the actual operation is performed on the input data. This extension of the Android Keystore, provided by Andromeda, is completely transparent to the developer. All necessary modifications are performed at the native C/C++ part of Android's Keystore while the corresponding Java API remains unmodified, rendering it completely backwards compatible with legacy applica-

tions. Persistent secure storage of keys and important metadata can be achieved using the SGX sealing technique. The Keystore service seals and exports the contents of the secure enclaves to the specified file system locations, protecting them during unexpected execution termination or device power-off. The exported data are encrypted and accompanied with the necessary metadata that ensure their validity. Once Keystore's enclaves need to be re-enabled, the service repopulates them by loading and unsealing the data. If the data is invalid or tampered, the service provides the necessary exceptions.

6.5.2 Native Development

Using the Andromeda SGX toolchain, developers can create their own SGX enclaves for their Android applications. To do so, native code in C/C++ has to be developed for the enclave functionality as well as the respective `ECALL`s and `OCALL`s that manipulate the data (sensitive or not) in the trusted and the untrusted part. To access the SGX code and functions, JNI bindings must be provided to the Java part of the APK to connect it with the native C/C++ and SGX counterparts. These JNI functions must be developed to initialize the enclave instance, setup the environment and access the secure enclave code, functions, and data. The process is quite similar to a Linux environment; the basic difference with SGX-enabled Android applications is that all native C/C++ code that implements the SGX enclaves and the native C/C++ code that handles their execution should be cross-compiled with the Andromeda toolchain which handles all the steps required to build the source tree.

6.5.3 Andromeda Java API

To assist the development of SGX-enabled Android applications, Andromeda also offers an API that developers can use to offload specific parts of the application into secure enclaves. The Andromeda Java API provides a set of building blocks for APKs and automates the generation process of secure enclaves that execute only minimal parts of the application logic in the trusted environment. The Andromeda Java API calls are shown in Table 6.1 and allow the creation of enclaves, the configuration of input and output between enclaves, and the execution of user-defined functions.

Secure Execution

The Java functions provided by the Andromeda API offer the following functionality. The developer can create a new secure enclave Java class instance using the `TrustedEnvironment()` constructor. To establish the trusted environment, the secure enclave Java class provides the `load()` method that passes configuration settings and user-defined configuration extensions to the enclave. This operation generates a new enclave using the C/C++

Table 6.1: Andromeda Java API for SGX enclave utilization.

Constructor Summary	
Constructor	**Description**
TrustedEnvironment()	Creates a new secure enclave class instance
Method Summary	
Modifier and Type	**Method Description**
void	**load**(EnclaveConfig config) Initializes the secure enclave
EnclaveOutput	**run**(int index, EnclaveInput i) Performs the trusted execution
int	**store**(byte[] data) Stores the data and returns its index
byte[]	**retrieve**(int index) Retrieves the data using its index
SealedData	**seal**(Object d) Seals the enclave data and stores to file system
Object	**unseal**(SealedData d) Unseals the data and populates the enclave
void	**pair**(ChannelConfig config) Creates a secure connection with the external device
void	**transmit**(ChannelConfig config, byte[] data) Securely transmits data to the external device
byte[]	**receive**(ChannelConfig config) Securely receives data from the external device
void	**terminate**() Disconnects the external
void	**destroy**() Destroys the secure enclave

layer of the Andromeda API and provides the necessary handles to the Java counterpart to interface with the enclave. The enclave and its metadata can be securely erased using the `destroy()` method, which optionally passes finalization data to the enclave. Developers can use the `run()` method to perform a trusted execution in the secure enclave. The `run()` method is extensible and includes the code that performs the desired computations inside the SGX enclave. Andromeda also provides the option to implement multiple functions to be executed in the trusted environment which can be invoked using their respective index (using the corresponding `run()` method argument). The `run()` method can be called an arbitrary number of times with different inputs.

In contrast to the manual development of SGX-enabled Android applications, when using the Andromeda Java API, the Andromeda toolchain generates the appropriate native C/C++ SGX code that implements the functionality defined in the `run()` method. Moreover, the toolchain generates the enclave driver code that handles I/O and function calling and establishes connection with the Java API by creating the necessary JNI bindings.

Secure Vault API

The Java functions provided by the Andromeda secure vault API enable both short term and persistent secure storage functionality. The developer can use the `store()` function to store a data object within a secure enclave. The data object can be of any kind, such as cryptographic keys, certificates, fingerprints, tokens or any other data considered sensitive in the scope of the application. Upon successful data storage, the API returns an index which can be used to retrieve the actual data through the `retrieve()` function. Moreover, the Andromeda Java API provides access to the SGX sealing and unsealing functionality, via the `seal()` and `unseal()` methods respectively. Using the `seal()` function, the developer can encrypt the data within the enclave using a secret key derived within SGX. Once the data are sealed, they can be stored in main memory or storage, with assurances for their integrity and authenticity, and can only be unsealed using `unseal()`. These functions can also be used to periodically generate backups of the secure storage to prevent data loss (e.g., from unpredictable execution termination).

Secure Pairing API

The secure device pairing functionality is provided by dedicated Andromeda API methods. These methods can be utilized by the Android application controlling the external device, as long as the external device includes Andromeda's connection libraries, which do not require SGX support, in its software stack. The developer is able to establish a secure communication channel with an external device using the `pair()` method. The external device can be connected either via Bluetooth or Wi-Fi. Andromeda then performs the attestation procedure for both devices. The configuration data passed to this method indicate the device ID, the attestation procedure (Remote Attestation or OTP), the option of notifying the user with a verification pop-up and other metadata, essential for initiating the connection. Once the attestation process is completed, Andromeda performs the communication channel establishment automatically, as described in Section 6.3.2. Once communication is initiated, the devices are able to exchange data using the `transmit()` and `receive()` functions respectively. Finally, the developer can execute the `terminate()` function for a `TrustedEnvironment` instance to securely disconnect the external device.

6.6 Evaluation

6.6.1 Security Analysis

In this section, we discuss Andromeda's security properties by describing possible attacks and showing how our proposed design protects against them.

Memory Attacks

We implement Andromeda in a way that nothing but a pointer to enclave memory is ever written into host memory. The pointer's content cannot be read or modified since it resides into the enclave. When Andromeda performs the desired operations, the output is transferred back to Android memory. In the meantime, we keep the enclave's execution alive and completely isolated from the Android system, without being affected by side effects of the OS or hardware, such as interrupt handling, scheduling, swapping, and ACPI suspend modes.

Controlling the Kernel

In cases where the attackers have successfully taken full control of the Android OS kernel, any sensitive data manipulated by Andromeda is still sound and safe. Once again, even though the attackers may have full *read/write/execute* rights in the whole system, they cannot read/write/execute code inside the enclave. As a result, any attempt to modify or read enclave code will result in a segmentation violation since this memory is not mappable outside the enclave, keeping the data secured.

Integrity of data

In a typical scenario, attackers can exploit software vulnerabilities and manage to inject code of their choice to a running service. Sensitive data, such as secret keys and checksums, stored in the address space of the process, can be easily acquired. In contrast, hiding sensitive data in a secure enclave prevents access even to fully privileged processes. To verify this, we attach our process with `gdb` to check the allocated pointers in the enclave code and trace the calls. However, no such data can be extracted since the enclave code and data are neither accessible from non-enclave code nor from the function calls or the memory stack. Such operations always result in segmentation violations.

6.6.2 Performance Analysis

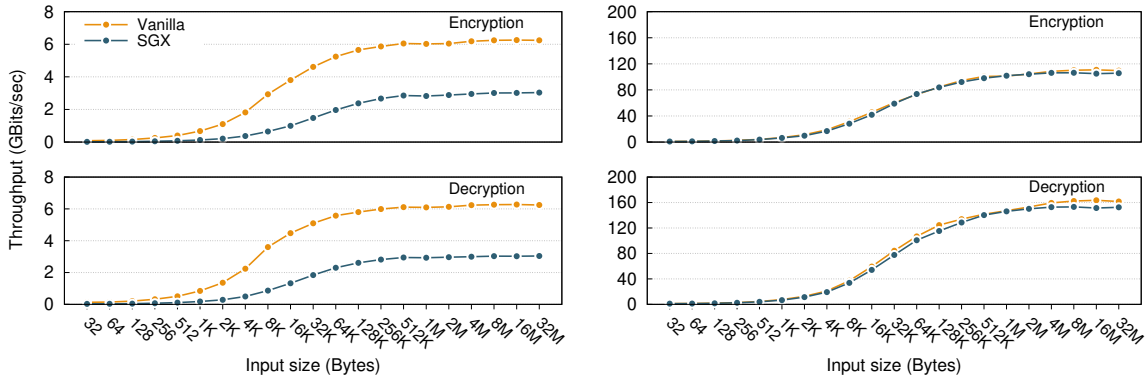
We now assess the performance of Andromeda and the extra overhead introduced for the execution of the secure enclaves.

Evaluation Setup

For our evaluation, we use an Intel NUC 8i5BEK kit with an SGX-enabled Intel i5-8259U CPU, clocked at 2.3GHz, and 8GB of DDR4 RAM. The system is running Andromeda OS (SGX-enabled Android x86 version 7.1.2_r33).

AES Evaluation

We compare the performance of the AES-CTR algorithm, as achieved by the vanilla Android Keystore system, versus the SGX-enabled implementation provided by Andromeda, using a custom benchmarking tool that we develop explicitly for this evaluation. In each processing loop, the tool generates a random secret key and a random stream of data. The data vary in size from 32B up to 32MB. To avoid any potential caching effects that may result in inaccurate results, we generate a new key and data stream in each processing loop. Once an AES key and a stream of data are prepared in memory, the tool performs cryptographic operations on the data using AES-128 in CTR mode, using both the vanilla and the SGX-enhanced Keystore system, provided by Andromeda. Figure 6.3(a) shows the performance characteristics of the native AES code execution. We achieve this by monitoring only the AES functions found in the native C/C++ part of the Android Keystore system. Our evaluation indicates that the overhead introduced by the SGX-enabled implementation ranges between 51% and 84% for the encryption operations and from 51% to 78% for decryption.



(a) Native Keystore level (b) APK level

Figure 6.3: Throughput comparison between the AES-128 CTR found in Android's Keystore and the SGX-enabled version provided by Andromeda's Keystore. Figure (a) presents the throughput achieved by Keystore's low-level functions while Figure (b) presents the throughput perceived at the APK level.

In the next experiment, we explore the throughput sustained in the APK scope. We achieve this by performing the same evaluation but in this case we monitor the execution time of the Java cryptographic functions provided to the APK by the Keystore system (Figure 6.3(b)). The execution time includes the entire execution path and the overhead introduced by the various layers of the Android architecture, including the IPC, the binder and the numerous function calls until the actual cryptographic operations are performed.

We notice that the sustained throughput perceived by the APK is one order of magnitude lower, compared to Figure 6.3(a), due to the overhead introduced by the various layers of the software stack involved in the process (i.e., JNI, IPC, and the binder). Similarly, the perceived overhead introduced by the SGX enclaves is minimised, ranging between 0.6% to 13% for encryption and 0.6% to 11% for decryption.

RSA Evaluation

We now present the performance comparison between the vanilla and our SGX-enabled implementation of the RSA algorithm. We perform the evaluation as follows. We develop a benchmarking application capable to perform RSA key generation, encryption and decryption. In each processing loop, the tool generates a new RSA key-pair and performs cryptographic operations against a set of input data. The data set consist of 10,000 random data chunks, varying in size from 32B up to 32KB, with each set containing chunks of the same size. We choose to generate a new set of random data in each processing loop to eliminate any caching effects. We execute the benchmarking application for every data set, each time monitoring the number of sustained cryptographic operations per second. The outcome of this evaluation is displayed in Figure 6.4.

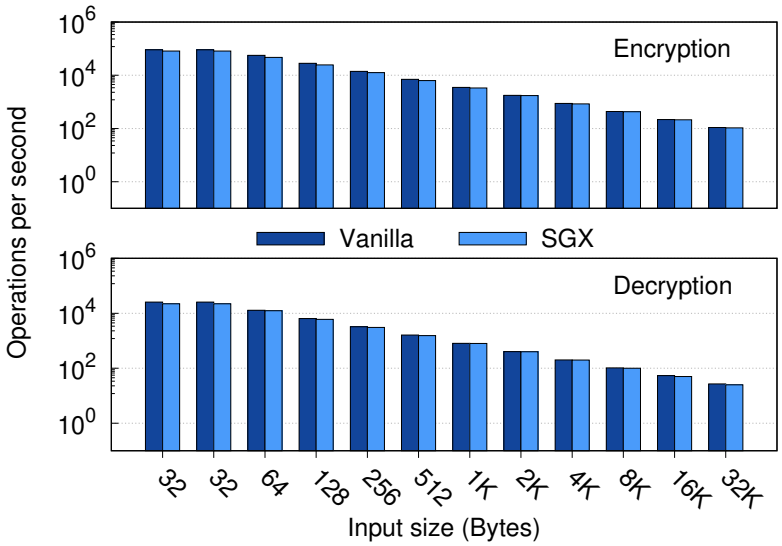


Figure 6.4: Sustained throughput achieved for the vanilla and the SGX-enabled implementation of the RSA-1024 cryptographic algorithm.

We notice that the SGX-enabled implementation introduces a maximum overhead of 16%, observed when processing a 64B input, with the lowest introduced overhead being 2.3% during the encryption of 2KB sized data blobs. The maximum sustained decryption

rate is observed for the vanilla implementation during the encryption of 32B data blobs with the introduced overhead being 12.6%. The minimum observed overhead introduced by the use of SGX enclaves is 0.9%, encountered during the decryption of 2KB cipher-texts. For both cryptographic operations, we observe that the perceived overhead introduced by the I/O between the benchmarking application and the SGX-enclave is minimised due to the processing complexity of the RSA algorithm.

Secure Device Communication Evaluation

In this section we present the performance evaluation of the data transmission between our central hub, an Andromeda-enabled device, and an IoT device, namely a Raspberry Pi. More specifically, we perform the evaluation as follows. We develop a benchmark application, able to utilise the secure device pairing described in Sections 6.3.2 and 6.5.3. The application generates and transmits data from the IoT device to the central hub with sizes varying from 32B to 32MB. At each transmission we measure the sustained throughput at the central hub. The measurements are performed twice, first measuring the achieved throughput using an unencrypted channel and secondly using end-to-end encryption, measuring the sustained throughput and latency introduced by the secure channel provided by our device pairing mechanism. We evaluate our system using two different wireless channels, a Wi-Fi 802.11g and a Bluetooth 4.0 connection. In the Wi-Fi setup, the IoT device and the central hub are interconnected via a public Wi-Fi access point while in the Bluetooth setup the two devices are connected directly. The reported values presented are the average of 100 executions.

Wi-Fi Evaluation The results of the Wi-Fi evaluation are displayed in Figure 6.5. The Y-axis represents the reported throughput in Mbps while the Y2-axis represents the per-byte latency introduced by the secure channel in nanoseconds. The X-axis represents the total amount of transmitted data. The maximum throughput achieved by our setup is measured at an average of 14 Mbps, using `iperf` [12], prior to performing any measurements. We observe that for small data streams, up to 2KBs, our secure channel introduces a maximum overhead of 7.5% in comparison to the unencrypted channel. The maximum achievable bandwidth is achieved with data sizes varying from 4KB to 32MB, with a maximum reported overhead of 3.8%, introduced by the data encryption and decryption processes at the sender and transmitter ends.

Finally, the latency evaluation reveals that for small data streams, up to 512 bytes, the initialisation time of the encryption and decryption processes introduce per-byte latency varying from 2937 to 183 nanoseconds. The introduced latency stabilizes for data streams larger than 1KB and averages at 38 nanoseconds per transmitted byte.

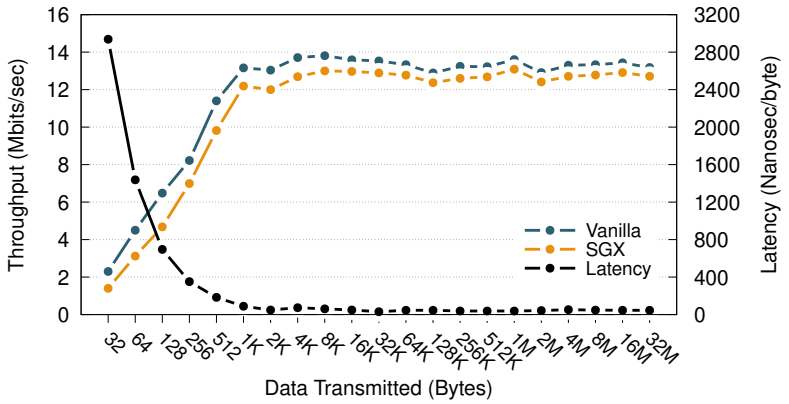


Figure 6.5: Sustained throughput of the vanilla and SGX-enabled data transfers between the host and the securely paired device using Wi-Fi.

Bluetooth Evaluation Figure 6.6 presents the evaluation results of the secure communication channel using Bluetooth. The X-axis represents the size of the transmitted data. The Y-axis represents the achievable throughput in Mpbs while the Y2-axis displays the per-byte latency introduced by the secure channel in nanoseconds. Prior to executing our benchmark application, we measure the maximum achievable throughput of our Bluetooth connection at 1.5 Mbps using a custom bandwidth measurement application, since, to our knowledge, no Bluetooth bandwidth benchmarks were available for Android.

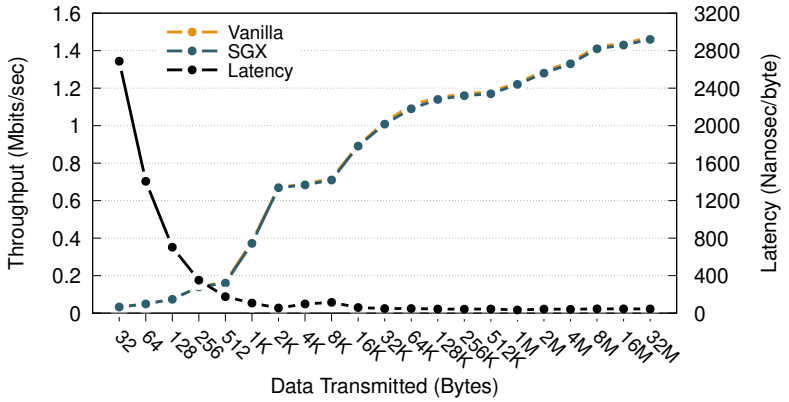


Figure 6.6: Sustained throughput of the vanilla and SGX-enabled data transfers between the host and the securely paired device using Bluetooth.

During our experiments, we notice that the Bluetooth connection is quite unstable and various environment variables, such as physical obstacles and distance between the devices, have a great impact on the achievable bandwidth. For data streams up to 32KB, our secure channel implementation, using Bluetooth, introduces a maximum overhead of 0.71%. For streams varying from 64KB to 8MB, the sustained throughput increases and gradually tends to stabilize at an average of 1.3 Mbits/sec, with an introduced overhead of 0.7%, caused by the end-to-end encryption. We observe that the per-byte latency is the same as reported during the Wi-Fi setup measurements, since the size of the encrypted and decrypted data is the same in both cases.

Computation Offloading

We conclude our evaluation by presenting the performance of three benchmarking applications, executed using the different methods provided by the Andromeda framework, as well as the overhead introduced by executing them remotely. In particular, we compare the execution of the vanilla Java implementations against their secure implementations using C/C++ and SGX natively, compiled with our custom cross-compiler, and their implementations using the Andromeda Java API for SGX. These benchmarks represent some typical operations that external devices or wearables may perform on sensitive data (e.g., analytics on finance or health data, image processing, etc.) and also exhibit different performance characteristics (i.e., I/O-bound, memory-intensive, computationally-intensive).

The first benchmark performs matrix multiplication on two tables with 10K rows and columns. The second performs bubble sort on an array of 20K random integers. Finally, the third benchmark is a convolutional neural network that performs image classification using as input images sized at 800x600 pixels, generated by an external device.

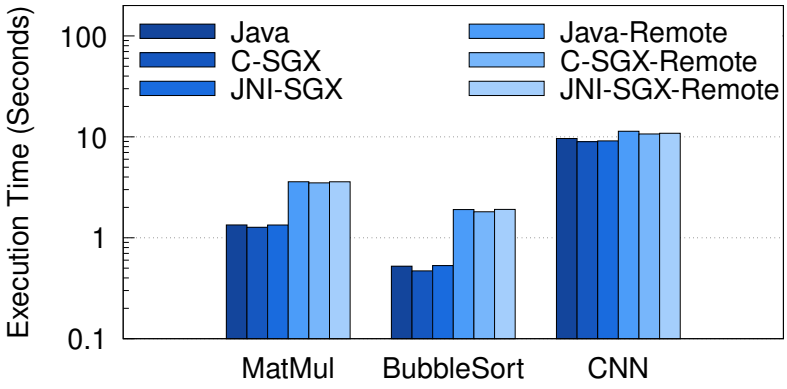


Figure 6.7: Performance comparison of the different Andromeda-enabled execution methods, including offloading, against the vanilla Java-based versions.

As we can see in Figure 6.7, the vanilla Java implementation requires 5.4% to 11.2% more time to finish its execution than the respective SGX-enabled implementations (developed either in native C or using the Andromeda SGX Java API) whereas the time needed for code offloading ranges from 5.13% to 7.5% for the Java-based implementation. The reason for this is that in both SGX-enabled versions, the functions are executed natively using C. The overheads introduced by the I/O with the secure enclave, the JNI layer (in the SGX-enabled Java implementation) and the data offloading on the socket level are minimal in these cases and do not overshadow the speedup gained by the native execution.

6.7 Summary

In this chapter, we presented the design, implementation, and evaluation of Andromeda, our framework's component that provides the first SGX interface for the Android OS. Using Andromeda, developers can explicitly use SGX enclaves for their applications via the native API, in C/C++, or via our Java interface that provides access to the secure enclaves through JNI bindings. Also, Andromeda offers services that enhance Android's security and provides protection schemes for applications that deal with sensitive data. The contributions of this work are the following:

- We present a systematic methodology for porting the SGX framework to Android, including the SGX kernel driver, the required libraries and its background services and the development of a custom cross-compiler. Android developers can explicitly use SGX for their applications either via the native API in C/C++, or via our Java interface.
- We implement popular Android services, enhanced with SGX capabilities, hence securing and protecting their functionalities. The SGX enclaves enable multiple secure spaces that can be used simultaneously by different applications, in contrast to other TEE ecosystems, such as ARM TrustZone, that allow only a single secure space that is shared among applications and often times requires control of the device's firmware.
- We implement a programming paradigm tailored for externally paired devices, that enables a robust, efficient, and trusted data flow between external devices that pair with the Android OS. Such devices can securely offload data storage and computations to the Android OS in a trustworthy manner, without necessarily being equipped with TEE-enabled CPUs.

As part of our future work, we plan to port Andromeda to SGX-compliant approaches that do not depend on specific CPU models, either using software-only techniques [83] or on top of ARM TrustZone [55]. Also, we plan to enhance our secure pairing mechanism by utilizing protocols that offer mutually trusted secure communication channels between enclaves that reside in different physical devices, similar to [64].

Chapter 7
Secure and Attested Communications

In this chapter, we present the design and implementation of our secure communication module that provides attested channel establishment for SGX-enabled devices where at least one of the communicating ends may reside in a potentially untrusted remote location, such as a third-party cloud platform. For this purpose, we build our system based on Intel's Remote Attestation process to enable secure key exchange and authorization of the communicating ends. Our system exposes a simple API with which two communicating entities can verify and attest each other, identifying if signed SGX-enclaves are used at both ends, running with hardware support, and create an SGX-to-SGX encrypted communication channel. Moreover, we implement a caching system for SGX Remote Attestation responses which greatly reduces the latency of new connections and benefits applications that require multiple short-term connections.

7.1 Intel SGX Remote Attestation

Remote attestation is the process of verifying the authenticity of a software component, running inside an isolated container, to some remote party. In the case of SGX, the software being attested is a secure enclave created by the trusted CPU hardware. For the remote attestation procedure, the CPU generates a measurement for the attested enclave which uniquely identifies it. This information is then signed by the privileged Quoting Enclave, resulting in an attestation signature (QUOTE). The Quoting Enclave is a special trusted enclave software, issued by Intel, and has access to the SGX hardware attestation key that signs the measurement. The attestation signature is generated using the EPID group signature scheme [104] to preserve privacy. The communication between the two enclaves must also be performed in a secure way. This is achieved by performing a local attestation between the two communicating enclaves as a means to establish a secure channel. The attestation signature can then be sent to the remote party, who relays this information to the Intel Attestation Service (IAS) to verify its validity. Thus, the remote party can be aware if the enclave has been tampered or if the attested software is not running

within a genuine hardware-assisted SGX enclave. This information is critical as it verifies that the SGX enclave is executed on SGX-enabled hardware and not in simulation mode, which renders the enclave accessible by debugging utilities. The SGX Remote Attestation process utilizes a modified SIGMA [113] protocol, therefore at the end of the process the remote party and the enclave establish a shared secret for secure communication. Contrary to Trusted Platform Modules (TPMs), SGX Remote Attestation has the benefit that attested software runs within the CPU thus having better performance. Moreover, SGX utilizes an EPID group signature scheme and attested enclaves cannot be uniquely linked back to a specific CPU through their attestation signature.

7.2 Threat Model

For this component, we assume a set of client nodes that may be distributed over different area networks or even different geographical areas. These nodes are connected over a public, not necessarily trusted, network, over which they can transmit and exchange data. We also assume that the client nodes may be compromised by a powerful adversary with full-privileged access or even access to the physical hardware, while the CPU is further equipped with at least SGX1 capabilities.

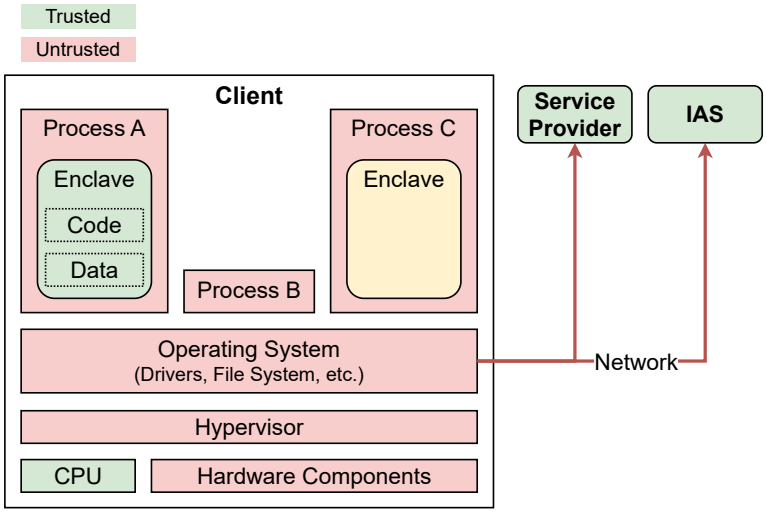


Figure 7.1: Assumed TCB and possibly malicious components.

The entities that are part of our TCB are the Intel SGX enclaves that can run in each client (excluding the container application and the hosting OS), the Service Provider, which acts as a directory server responsible for resource and user discovery, and the IAS, that verifies that the SGX hardware is genuine. Any other components, such as the host application and the underlying OS are not part of this module's TCB. An overview of our TCB and possibly malicious components is presented in Figure 7.1. Overall, our primary aim is to

defend against adversaries that can: (i) control and tamper applications and the client's operating system, (ii) observe and tamper data transmitted over the network channel, (iii) conduct man-in-the-middle attacks between either a client and the SP or between two client enclaves, and, (iv) perform replay attacks that utilize information from previous sessions. We note that our threat model excludes denial-of-service (DoS) attacks on enclaves since the life cycle of the process containing the enclave can be controlled by a malicious operating system or superuser. While adversaries can prevent or abort the execution of enclaves, they are not able to obtain any valuable information by doing so. Furthermore, side-channel attacks that exploit page faults or timing information are excluded from this work, even though we assume a small and well audited TCB. Finally, we assume that the design and implementation of the Intel SGX framework is free of vulnerabilities.

7.3 Design

7.3.1 Involved Parties

Our protocol involves three parties in total, namely (i) the *clients*, (ii) the *service provider*, and (iii) the *attestation service*. Each entity is shortly described bellow.

Clients The remote parties that want to communicate and support Intel SGX enclaves. For simplicity, in the rest of this chapter, we assume only two clients, Alice and Bob, where Alice wants to communicate with Bob via a secure communication channel, using Intel SGX. We notice though that our design can scale by nature and operate independent of the number of connected peers.

Service Provider (SP) The application's vendor who signs and ships the clients' SGX enclaves. Signing verifies that the code cannot be changed, tampered, or altered. Service Providers must register themselves with the Intel Attestation Service to make use of the provided services. To do so, the Service Providers must fulfill a set of standard requirements in order to submit their attestation results to the attestation service and verify that the system is sound. Consequently, this process assigns a TLS certificate to the Service Provider ID (SPID), thus granting access to the attestation services.

Intel Attestation Service (Intel Attestation Service (IAS)) The IAS is used to verify the attestation evidence that the Quoting Enclave generates and reports it back to the Service Provider. The Quoting Enclave (QE) is provided by Intel and its goal is to receive and verify reports from other enclaves, which converts and signs using the Intel EPID Key, which is a device specific asymmetric key. The attestation flow is the following:

1. The application receives an attestation request (REPORT) from a third party and forwards it to its enclave.
2. The enclave sends the local attestation request to the Quoting Enclave.
3. The Quoting Enclave validates the request and transforms it in a remote attestation request.
4. The request is returned to the application which forwards it to the challenger.
5. The challenger uses the Attestation Verification service to perform the verification.

7.3.2 Protocol Design

In this section, we present our protocol in detail, describing the messages exchanged between two clients, namely Alice and Bob. In our setup, we assume that Alice wants to initiate the communication. A graphical representation of the message flow is depicted in Figure 7.2. More specifically, the actions performed by the two parties are the following:

1. Alice requests communication with Bob from the Service Provider.
2. The Service Provider challenges Alice with a message containing a `nonce`.
3. Alice attests to the Service Provider by sending her generated `QUOTE`. The Service Provider sends the `QUOTE` to the IAS and checks the response. By doing so, the Service Provider verifies that Alice utilizes an untampered enclave, running on genuine Intel SGX hardware. In this step, the Service Provider also checks the key used to sign the enclave (`MRSIGNER`), the hash value of the enclave (`MRENCLAVE`), the software version of the enclave (`ISV_SVN`) and the enclave's product ID (`ISV_ PROD_ID`), which allows multiple enclave instances to be distinguishable. Intel [36], Hile Vill [195] and others [67] have thoroughly described this remote attestation process.
4. If the attestation is successful, the Service Provider challenges Bob.
5. Bob performs the same attestation process as Alice.
6. If the attestation is successful, the Service Provider notifies Bob that Alice wants to communicate with him. Then, the SP generates a shared secret for the establishment of the secure enclave-to-enclave encrypted channel between the two clients. For each client, the secret is distributed via a secure channel that has its client-side endpoint within the enclave and has been established during the IAS attestation step.

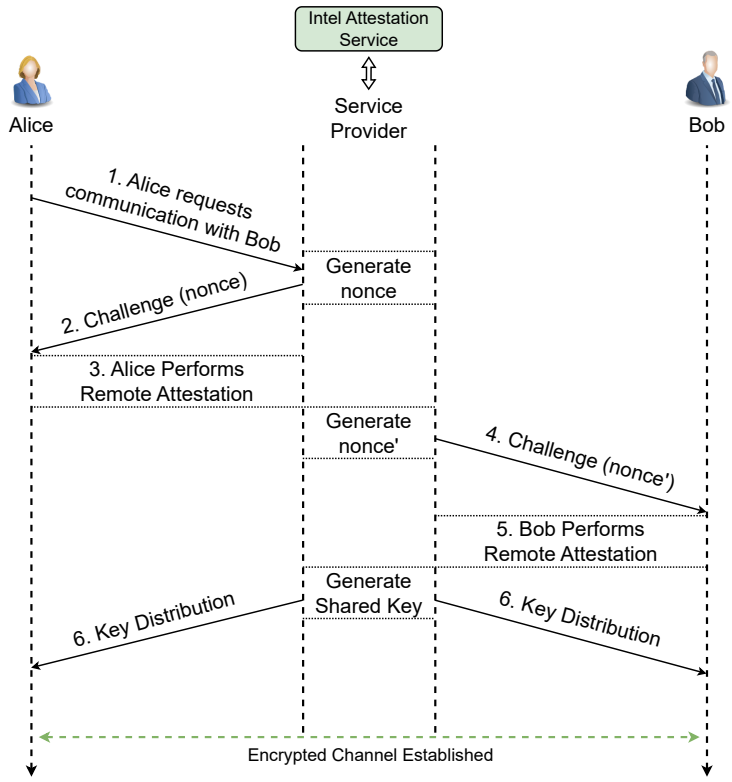


Figure 7.2: Attestation protocol.

After completing the aforementioned steps, the enclave-to-enclave channel between the two clients is established and Alice can communicate securely with Bob. The shared secret is used to guarantee the confidentiality, integrity and authenticity between the two enclaves and is never exposed outside of the SGX environment.

The protocol follows a serial approach and we leave its parallelization as part of our future work. Additionally, we assume that Alice does not know how to directly communicate with Bob and that Bob does not trust anyone that has not been verified by the Service Provider. This introduces a level of centralization which we discuss in Section 7.3.4.

7.3.3 Attestation Caching

The remote attestation procedure, as shown in Figure 7.2, consists of multiple stages across different entities and, as a consequence, it may require a significant amount of time to complete. These times are accumulated when applications require to communicate with multiple enclaves at the same time, since they would have to attest to the SP for each individual session. To make matters worse, the overhead of session establishment becomes even more notable for short-term sessions, in which the time required for the initiation of the session can be significantly larger than the overall communication time itself.

To reduce the remote attestation overheads, we extend our basic protocol, described in Section 7.3.2, introducing a caching mechanism at the Service Provider. The mechanism is responsible for caching the remote attestation results produced by IAS, aiming to increase the protocol's performance. Our approach also allows the fine-tuning of the caching period, based on the security requirements of the enclave that the Service Provider is attesting. During protocol initiation, the enclaves inform the Service Provider about their intent to use cached attestation results. However, the SP is the final arbiter of whether or not a full remote attestation should take place. We implement two caching approaches:

1. TTL-based ephemeral session keys In this approach, the Service Provider and the client enclaves store the session keys in memory for future use. This is a safe operation since the enclave's memory is encrypted, checked for integrity violations, and is also inaccessible from the untrusted environment. The Service Provider can define a Time-To-Live (TTL) for which the keys are assumed to be valid before requiring the client enclave to perform a complete remote attestation again. In each attestation attempt, the SP challenges the enclave with a `nonce` and expects the correct `HMAC` value as a response.

2. Session key hashes During the first protocol instantiation with the client enclaves, the Service Provider uses a secure hash function H to store the hash value of a session key, $H(KEY_{\mathrm{SK}})$, as opposed to storing actual keys in the previous approach. Afterwards, each time the client's enclave attests to the Service Provider, the SP issues a `nonce` to the enclave and expects it to respond with the correct $H(H(KEY_{\mathrm{SK}})||nonce)$ value. This method is similar to remote user authentication using passwords [178] but since the keys are ephemeral there is no need for salt. The stored hash values can be associated with a TTL field or a fixed number of maximum N sessions to be initiated using the same ephemeral key.

Both approaches can be applied by the Service Provider, meaning that different enclaves that attest to the Service Provider can follow different approaches. Additionally for the implementation of the caching mechanism, the Service Provider can decide on which key will be used as caching key. This can be achieved in two different ways:

1. Enclave session key caching The Service Provider uses as caching key the session key established with the attested enclave during the remote attestation with the IAS in order to implement caching for that particular enclave.

2. Enclave-to-Enclave channel shared secret caching The Service Provider uses as caching key the shared secret generated by the Service Provider during the establishment of the secure communication channel between the two enclaves in order to implement caching for that pair of enclaves.

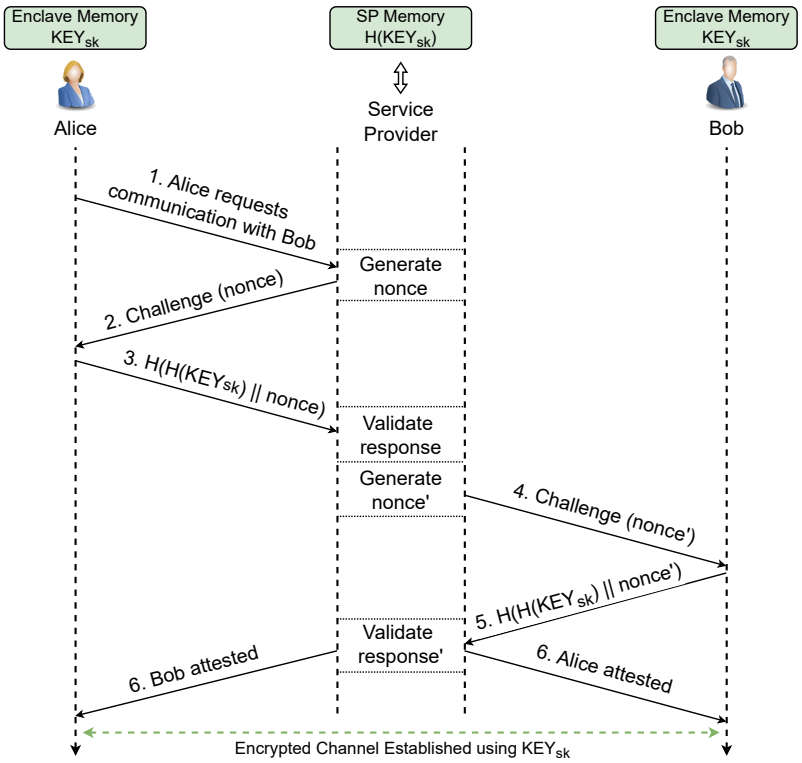


Figure 7.3: Attestation protocol with response caching.

It is important to note that the Service Provider can force the enclaves to perform a complete remote attestation at any protocol instance or even opt-out of caching, depending on the security requirements and access control policies of the enclaves. Furthermore, when attestation caching is used, the IAS is not involved. This reduces the privacy footprint of an enclave regarding how frequently it attests to Intel. A graphical representation of the protocol using remote attestation caching with *Session key hashes* and *Enclave-to-Enclave channel shared secret caching*, is presented in Figure 7.3.

7.3.4 Protocol Centralization

During our research of the enclave-to-enclave communication problem, we came across the option of either including an intermediary remote party, namely the Service Provider, or implement a completely decentralized approach. We chose to include the Service Provider because it offers the following benefits:

- Remote Attestation is a complex procedure involving trust policies which can vary based on the required security and strictness of the application. These policies become even harder to define when there are multiple and different enclaves involved.

The Service Provider can dynamically change those policies without any modification to the enclave code or redeploying them.

- Different policies for different enclaves can be established. The Service Provider can allow different types of enclaves communicating with each other and then control the various trust levels between them.

- Only the Service Provider knows the communicating parties (Alice and Bob) and the protocol does not depend on other peer discovery primitives which can have privacy implications.

- We can assume that an entity with the role of the Service Provider exists as enclaves are signed before being deployed in a production environment. Any enclave that is to be provisioned secrets from the network should always be attested, thus raising the need for such a remote party. Moreover, since enclaves cannot be tampered, techniques like certificate pinning in combination with Enclave Signature Revocation Lists [195] can enhance the security even further.

- The Service Provider does not have to be a single point of failure as many instances can coexist. The remote party acts mostly as a policy enforcing entity and coordinator between communications. Also, the Service Provider(s) can also offer custom attestation infrastructure [163] to replace the IAS and mitigate cases where SGX's EPID and attestation report do not hold onto their claimed anonymity and privacy guarantees [163].

7.4 Evaluation

7.4.1 Evaluation Setup

We implement the clients and the Service Provider utilizing our proposed protocol using C/C++ and the Intel SGX SDK v2.6. Also, we use OpenSSL version 1.1.0, build on top of the SGX framework to perform all the required cryptographic operations. The overall system consists of 7290 lines of code. Three different machines are used to evaluate and test the implementation of our protocol, involving two clients and one SP. The two clients run Arch Linux with LTS kernel version 4.19 and utilize an Intel Core i7-8700K processor clocked at 3.70GHz along with 32GB of DDR4 memory. The SP is hosted by an Ubuntu 18.04 system, running Linux kernel version 5.0.0-36-generic, utilizing an Intel Core i7-8565U CPU clocked at 1.80GHz along with 16GB of DDR4 memory. All the enclaves and the SGX related binaries are built and executed in SGX1 Hardware Mode to take advantage of the SGX capabilities.

7.4.2 Performance Evaluation

For the evaluation of our protocol, we measure the time required for establishing a secure enclave-to-enclave communication channel between two clients, using our custom RA approach. Then, we compare the results to the time required to perform a standard TLS handshake with a valid certificate signed by an Intermediate Certificate Authority (CA).

We perform 200 TLS handshakes and measure the time required for the client to establish the TLS connection with the server. Also, we perform an additional 400 secure channel establishments using our Remote Attestation protocol, out of which half are performed without caching while the other half are performed with *Session key hashes* and *Enclave-to-Enclave channel shared secret caching*.

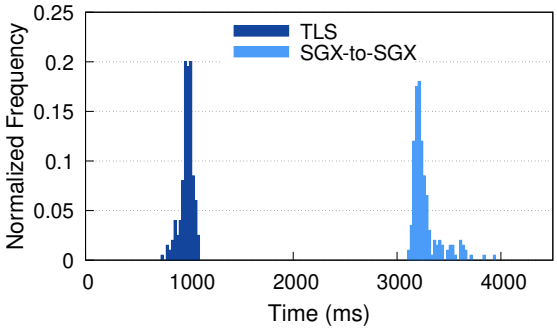


Figure 7.4: Execution time frequencies.

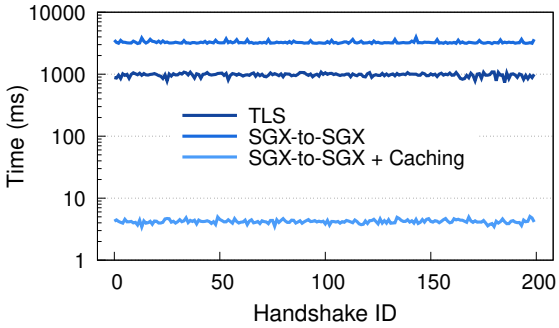


Figure 7.5: Execution time comparison.

As seen in Figures 7.4 and 7.5, our protocol, when no caching schemes are utilized, is approximately 3.5 times slower compared to the TLS handshake. We also observe that the standard deviation for the TLS handshake is 62.15ms while for our protocol it is 138.03ms. This is due to the fact that our protocol relies on the IAS, meaning that the time required for communicating with the IAS over the network and performing the remote attestation is also measured. However, as presented in Figure 7.5, we observe that our approach, when caching is deployed, is three orders of magnitude faster than the TLS handshake, with mean establishment time of 4.20ms and standard deviation of 0.27ms. These time statistics are more thoroughly presented in Table 7.1.

Table 7.1: Execution time statistics in milliseconds.

	TLS Handshake	**Remote Attestation (without caching)**	**Remote Attestation (with caching)**
Average	974.93	3269.66	4.2
Standard Deviation	62.15	138.03	0.27
Minimum	733.07	3113.54	3.43
Maximum	1088.67	3940.94	5.06

7.5 Summary

In this chapter, we introduced our secure communication protocol leveraging the Remote Attestation feature provided by the Intel SGX framework to build mutually trusted secure communication channels between two enclaves, residing in different physical machines. Furthermore, we proposed a caching scheme for the remote attestation results that does not compromise the security and privacy of our model and speeds up consecutive remote attestation processes. Finally, we evaluated our system by comparing the time required for establishing a secure channel between two enclaves using our approach to the time required for a TLS handshake to be completed. Our results show that our system has 3.5 times more overhead compared to the TLS protocol when we do not utilize any caching schemes. However, our protocol instantiation has substantially increased performance when attestation caching is applied, rendering it faster than a standard TLS handshake. This work's contributions are the following:

- We design and implement a protocol that provides remote attestation of communicating ends executing inside SGX enclaves.
- We extend our framework with a module that provides seamless establishment of enclave-to-enclave (CPU-to-CPU) encrypted network communication between two or more parties.
- We propose a caching system for SGX remote attestation responses that reduces the latency of consecutive connections, rendering them substantially faster than a standard TLS handshake.
- We provide a comparison of our secure communication method against commonly used methods such as TLS.

In the future, we plan to explore the performance and privacy benefits of entirely excluding Intel's infrastructure from the loop and utilizing our own infrastructure at every protocol step, as enabled by the latest SGX2 capabilities. Also, we aim to parallelize our protocol to further improve is performance.

Chapter 8
Trusted Execution of High-level Dynamic Languages

This chapter presents LuaGuardia, a system that aims to simplify the development of confidential computing on Trusted Execution Environments. LuaGuardia offers a set of high-level abstractions for building applications for, or porting applications to, TEEs. LuaGuardia's key insight is to embedded the runtime environment of a high-level programming language into the TEE — in this case, the language is Lua and the TEE is Intel SGX. We choose Lua due to its high-level semantics, its dynamic meta-programming facilities (which include runtime code evaluation), and its small memory footprint; the LuaVM consists of ~ 24,000 LoC, consuming minimal enclave memory and thus leaving more memory resources available for the applications running atop LuaGuardia. Our system packages a small runtime library for dealing with application signing, extensibility, system call forwarding, and other technical challenges.

We apply LuaGuardia to a large set of programs, among which are three large and high-performant applications, ported to leverage TEE facilities: (i) an HTTP telemetry tool that TEE-offloads its sensitive analytics, (ii) a packet-filtering tool that TEE-offloads its packet matching facilities, and (iii) 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 introduced by the TEE utilization ranges between 13% and 41% (avg: 18%). Applying several optimizations significantly lowers the introduced overheads, especially for short-lived program fragments.

8.1 TEE Computing Challenges

Developing a program to execute on a TEE poses several challenges. To make these challenges concrete, we describe a set of example applications, 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.

Let's consider a few confidential computing scenarios where part of an application needs to run on untrusted but 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 system 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 application part 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, causing several practical challenges.

1. 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 APIs, it still requires manual partitioning, recompilation, library porting, and linking of the entire application, including the components running out of the TEE.

2. 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 is impossible. In these cases, adding even a single-function module requires compilation, linking, and redeployment 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.

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

8.2 Addressing the Challenges

LuaGuardia embeds a Lua runtime inside a TEE to address the challenges of: (i) manual effort, (ii) dynamic extensibility, and (iii) runtime safety, by piggy-backing on the char-

acteristics 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 [102].

1. Manual Effort LuaGuardia offers significant development 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. Since 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, as we show this in our evaluation section.

2. Extensibility Based on a dynamic language, LuaGuardia is naturally extensible via runtime 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.

3. Safety Finally, as Lua is a memory- and type-safe programming language, the engineer does not need to worry about low-level attacks in the 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 memory-safety guarantees provided by the Lua environment.

8.3 Threat Model

LuaGuardia assumes a powerful malicious party, possessing root privileges, who also has physical access to the hardware but cannot physical tamper with the CPU. Moreover, the adversary is able to manipulate the entire software stack, including the operating system kernel. However, denial of service attacks (DoS) on key components, including but not limited to blacklisting the SGX or NIC drivers, preventing access to the file system etc., are out of the scope of this work. Since a malicious party manipulating the operating system is able to control an enclave's life cycle, we assume that they are able to prevent or abort the execution of the SGX enclaves or disrupt network communications but they should not be able to gain any useful information by doing so.

Moreover, side-channel or other types of attacks targeting the Intel SGX hardware components, such as timing or page fault exploitations, are orthogonal to LuaGuardia and any work that successfully manages to mitigate such attacks on SGX in the future can have a

direct benefit to our system. Furthermore, LuaGuardia assumes that the design and implementation of the Intel SGX framework is free of vulnerabilities. Finally, we assume that the implementation of the enclave-protected code as well as its untrusted driver-code are free of software vulnerabilities or bugs that could compromise them via remote input.

8.4 Architecture

The architecture of LuaGuardia is shown in Figure 8.1. The system provides two execution entities: the *untrusted*, where the request handling is managed, and the *trusted*, where the LuaGuardia protected LuaVM resides along with any sensitive data. LuaGuardia provides the necessary secure communication mechanisms between the two entities. Typically, an application that runs in the untrusted domain aims to offload security-sensitive computations to the trusted component. The latter is responsible for accepting such requests and performing secure code execution, encapsulated within SGX enclaves.

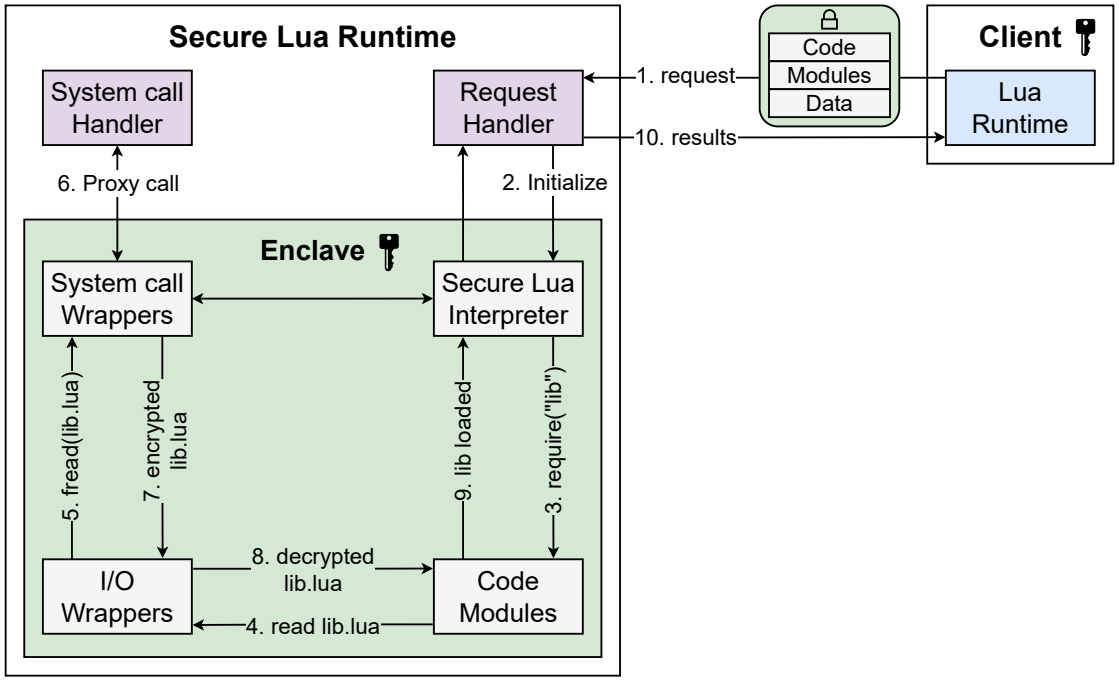


Figure 8.1: LuaGuardia architecture overview and secure execution life cycle.

8.4.1 Interpreter Enclaves

In the trusted environment, LuaGuardia provides a language interpreter environment, based on Lua. It mainly consists of an SGX-enabled LuaVM, responsible for performing

the code execution entirely within SGX enclaves. The code running in LuaGuardia can only perform computations and has no access to system calls or peripherals. The handling of the incoming client connections, as well as the serving of the required system calls and I/O is the responsibility of the enclave driving code. By doing so, we limit our system's TCB while being able to offer enough functionality to meet the security requirements of many applications. In Section 8.6 we evaluate three such representative real-world applications.

The execution circle of the LuaGuardia server is the following. Initially, a client that requires to perform secure Lua code execution has to perform a remote execution request to the server, providing its public key (*step 1*). The server stores the client's public key inside the enclave and performs a session key exchange, entirely inside the SGX enclave. The posted and generated cryptographic keys are never exposed to the untrusted part of the server. Once the session key is generated and exchanged with the client, the two entities establish a client-to-server-enclave communication channel that prevents data leakage outside of the enclave. Then, the server receives the offloaded Lua code from the client along with all the required modules and input data in an encrypted format, unusable outside of the enclave (*step 2*). The untrusted part of the LuaGuardia server provides the encrypted data into the enclave where they are decrypted. Once all the required Lua scripts and data are decrypted in the enclave, they are loaded into the protected LuaVM where they are executed (*steps 3-10*). The generated results are then encrypted with the session key and then exported to the untrusted part of the server, responsible for transmitting them back to the client.

8.4.2 LuaGuardia Client Stub

The LuaGuardia client stub is designed as a thin library. It enables applications to securely connect with the LuaGuardia server, over a client-to-server-enclave encrypted communication channel, and request the execution of a Lua script in a protected memory space. The library-based design enables new and legacy applications to easily utilize the remote service with minor modifications. Moreover, the LuaGuardia client does not require SGX capabilities and its only crucial operation is to establish the secure communication channel with the server, using commodity cryptographic operations that do not require special ISA extensions or acceleration. This enables the client library to operate even on low-power mobile or embedded devices.

Every time an application requires secure Lua code execution, it performs the following steps. First, it generates a new key-pair and posts its public key to the remote server, requesting, dynamically, a new code execution environment (*step 1*). Once the server is informed about the request, both entities perform a key exchange. On the client side, the key establishment is performed in untrusted memory since we assume that the requesting application is executed on a host that does not provide a TEE, although a TPM can

be used for that matter. On the server side, the communication channel establishment is entirely performed inside the server's SGX enclaves. Moreover, the cryptographic keys associated with the new request as well as any generated session keys are always secured in the server's trusted memory space and are never stored in the untrusted DRAM or the file system. This process generates a client-to-server-enclave communication channel that prevents malicious eavesdroppers, monitoring the network or the remote server's memory and file system, from obtaining the exchanged data.

Once the communication channel is established and the session key is exchanged, the client packs the Lua code that needs to be securely executed along with its required Lua modules and input data, and encrypts them using the newly generated session key. Then, it transmits the encrypted data to the server and awaits for the result (*steps 2-10*).

8.4.3 Local Execution

The LuaGuardia server optionally supports a local execution mode where the client and the server are co-located into the same physical host. This option enables newly developed or legacy applications to utilize SGX to securely execute Lua scripts without the need to transfer the code and its required data over the network. We note that this mode can be used only under certain conditions. More specifically, this operation mode can be considered successful only if we assume that the malicious party cannot control the operating system or the file system, while other user space applications can be compromised, even completely. The reason for this is that both code and data will reside in plain-text format in untrusted memory or in file system locations before they are loaded into the secure LuaVM. However, this follows SGX's standard, local, execution model. Performing code and data encryption in this scenario is meaningless since this operation will take place in unprotected memory. In such cases, an adversary controlling the file system can tamper the Lua scripts or their input data prior to their execution or whilst they are being transferred into the SGX enclave.

8.5 Implementation

In this section, we present the implementation details of LuaGuardia as well as the challenges we encountered while porting the LuaVM into SGX enclaves.

8.5.1 Porting the LuaVM

LuaGuardia is based on Lua version 5.3.5 [13]. Compiling the vanilla Lua project from source generates two distinct binaries. The first one, named `lua`, is the standalone Lua interpreter, able to load and execute Lua scripts either is source code or pre-compiled binary format. The second one, namely `luac`, 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 choose to utilize the official Lua interpreter instead of the third-party developed LuaJIT [143], despite the latter offering considerable performance increase, for several reasons. First, using the official interpreter guaranties that all language features will always be up to date, according to its standards. Second, the LuaJIT project has a bigger TCB, a property that we wish to keep as small as possible for memory efficiency and auditing reasons. Finally, SGX1 enclaves do not support the creation of additional executable memory pages during execution.

Our LuaGuardia implementation utilizes the Lua interpreter instead of the Lua compiler as the former is the one that provides the execution environment required to execute either Lua source code scripts or pre-compiled Lua binaries. The vanilla Lua interpreter codebase utilized for the development of LuaGuardia consists of 25 header files, containing 4269 LoC, and 35 source code files containing 19482 LoC in native C. The total codebase spans 23751 LoC, rendering the Lua interpreter a perfect candidate for a protected dynamic language environment as it has a small TCB that can be easily audited.

We port the Lua interpreter into Intel SGX enclaves on an Arch Linux based host, running kernel version 5.7.6, using the Intel SGX SDK version 2.8 for Linux and GCC version 9.2.1. The 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 SGX enclaves.

While the interpreter has a quite minimal codebase, the implementation of the LuaVM with SGX support is not straightforward. First, several modifications to the original source have to be made since `signals` and the `dl*` family of functions are not supported by SGX. Moreover, supporting them could potentially compromise the integrity of the enclaves. For these reasons, we make the appropriate modifications to disable or replace their functionality without affecting the proper operation of the protected Lua interpreter. Also, several function calls resulting in system calls, such as `fopen`, `fwrite`, `clock`, `ops` etc., cannot be executed as SGX enclaves do not support system calls. For this reason, we develop wrapper functions to proxy them to the untrusted part of LuaGuardia, where they are served as SGX's model dictates.

Apart from providing a secure SGX port of the Lua interpreter, we also have to develop the appropriate functionality so that LuaGuardia can operate as a remote service, able to establish secure communication channels with the candidate clients. Moreover, the server has to be able to handle and parse encrypted Lua code blobs which are going to be executed within the protected Lua interpreter. Finally, we develop a protected Lua interpreter instance manager, responsible for handling the various clients and provide each one with a separate instance for isolation purposes.

8.5.2 System Call Handling

A common issue with many TEEs, including Intel SGX, is the lack of system call support. Given that only the software that runs inside the TEE can be considered trusted, any calls to components that reside outside, including the operating system kernel or the hypervisor, can pose security risks. A direct system call from an SGX enclave may compromise and expose enclave code as well as protected data, thus rendering its security scheme useless. However, almost all SGX-enabled applications need to invoke system calls to perform basic functionalities, such as access to the network or the file system. When utilizing Intel SGX, these calls are usually requested from within the enclave and are forwarded to the untrusted part of the application where they are handled. In essence, it is up to the developer to design and implement the appropriate intermediate software layer that serves the system call requests and hands their result back to the trusted enclave.

While a custom built SGX-enabled application might require only a few system calls, which can be easily wrapped by the developers, this is not the case for our system. LuaGuardia needs to provide a full-scale Lua virtual machine, enclosed in SGX enclaves, which introduces two challenges that need to be addressed. First, we need to handle the system calls required by the LuaVM itself in order to properly operate. Second, new system call requests can be issued with each different executed Lua script. The former case can be considered more straightforward, as the required system calls can be easily accounted for, which unfortunately is not the case of the latter, that cannot be known a priori.

The straw man approach for handling system calls is to implement custom wrappers for each and every one of them. However, modern operating systems provide several hundreds of system calls, about 400 in Linux, without necessarily being called by the LuaVM or an executed script. Moreover, many of these system calls could be triggered by an offloaded script and abused to perform malicious operations on the remote host. For this reason, we decide to provide interfaces only for the bare minimum number of functions required for the proper execution of the LuaVM inside the SGX enclave, aiming to minimize the potential attack surface without providing interfaces to the system calls directly.

The most commonly used functions, resulting to system calls, are the those associated with data I/O such as `fopen`, `fwrite`, `fread`, etc. To provide system call support to the LuaVM residing in the SGX enclaves, we have to develop custom functions that proxy the system call requests to the untrusted part of the application, performing one `OCALL` for each request. Once a request reaches the untrusted part, the appropriate system call is issued and served, and the results are forwarded back into the enclave. According to SGX's programming model, as soon as the execution transfers between the trusted boundaries, it is the programmer's responsibility to validate and verify the content of the returned results.

Since we do not trust any other entity other than the enclave, all data required for the execution of the offloaded Lua scripts must be provided with the client request, in en-

Table 8.1: Functions issuing system calls, ported to the enclave-protected Lua runtime.

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

crypted format, and we prohibit read and write operations to arbitrary file system locations at the server side. Optional file offloading and handling can be performed by the client in predefined server-side file system locations in an encrypted manner, using SGX's sealing and unsealing functionality, enabling secure persistent storage in the untrusted file system. A list of the supported functions resulting in system calls available to the protected LuaVM are presented in Table 8.1.

8.5.3 External Modules

By design, Lua is an extensible programming language that offers dynamic module loading capabilities. It has built-in support for two types of libraries: (i) libraries that are implemented entirely in Lua code, and (ii) dynamic libraries written in native C/C++ that can be registered through the native Lua C API. To support the former, we first need to implement the system call proxy layer described in the previous section. Second, we integrate the `FILE` data structure in the enclave's codebase as it is not originally available. With these two infrastructures in place, the LuaVM can read the module's code through LuaGuardia's untrusted part and perform Lua module loading without any other modifications. However, as previously described, proxying a `read` system call to the untrusted part does not guaranty data validity and, at this point, LuaGuardia is not able to verify if a target module is tampered by a malicious party. To lift this limitation, we further modify LuaVM's I/O functions responsible for module loading in the following way. LuaGuardia expects that a Lua module bound for loading will be read in an encrypted format. In this way, the untrusted part issues the `read` system call and delivers the encrypted module in the enclave. Afterwards, the module is decrypted, inside the SGX enclave, and its integrity is verified via a known checksum. If the checksum is correctly verified, the protected LuaVM loads the module. This design prevents attackers 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 can either be pre-stored in LuaGuardia's enclaves, such as checksums for standard libraries, or received in encrypted format by the client wishing to load the module. With this mechanism in place, any `require(<module_name>)` snippets found in a Lua script can import the declared library, assuming it is available, and the module's API is exposed to the developer.

Native C/C++ libraries, compiled as shared objects, can only be loaded in LuaGuardia's

untrusted part, using the `dl*` family of functions. This property prevents us from providing dynamic C/C++ library support for the following reasons. First, since shared objects cannot be loaded in LuaGuardia's enclave, the system is not able to verify their integrity. Second, the functions loaded in LuaGuardia's untrusted part can only be interfaced by the enclave-protected code via a function proxy layer, similar to the one described for serving system calls. However, the implementation of this layer requires knowledge of each function's prototype, something that is not possible for functions found in dynamic libraries.

8.5.4 Maintaining Global State

Typically, the lifespan of a trusted execution within LuaGuardia, consists of the following phases: (i) the setup of the trusted execution environment, including attestation and key establishment, (ii) the setup of the code and the necessary data, (iii) the subsequent code execution within the enclave, (iv) the transmission of the execution results back to the user, and (v) the termination of the session and the wipe-out of any program state or data.

The steps above enable a functional execution paradigm, without requiring to keep any state across program executions. By the end of each session, any state has been wiped out and the execution environment is dynamically reset to handle new code execution requests. It could be useful though for users to be able to export the given snapshot of the stack/heap, and start a new execution on a different Lua State. To accommodate such cases, we have added builtin support to LuaGuardia that enables users to preserve and restore the context of the stack/heap on demand, at any given execution.

More specifically, LuaGuardia makes use of the `environment` table that is maintained by the LuaVM to hold information and metadata regarding all the global variables of the current Lua session. At each new Lua session, a new `environment` is assigned to it and it is stored inside the `_G` global variable. The entries of `_G` can be accessed and iterated normally, thus injecting and restoring data is also feasible. LuaGuardia uses these mechanisms to export the current heap/stack state of the variables, as well as preserve all the function code and data at any given moment of the execution, and restore the state at a completely different Lua session/context. This is achieved by a special set of functions, provided by LuaGuardia's API: (i) the `LuaGuardia_dump`, which extracts the current state of the global variables from `_G`, extracts the code/functions and writes them in JSON format to a file, and (ii) the `LuaGuardia_parse` which takes as input a file in JSON format, created by the `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 later validation. Consequently, `LuaGuardia_parse` decrypts and performs integrity checks on the exported data prior to restoring them in `_G`.

8.5.5 Code and Client Isolation

By utilizing SGX enclaves, LuaGuardia guaranties that the protected LuaVM is isolated from the rest of the system, including other applications and the operating system kernel. To further guaranty client isolation, LuaGuardia registers a new protected LuaVM, residing in a separate enclave, for each client. Moreover, each client session is established with a different unique secret key that can optionally be preserved across different sessions with the same client. In this way, attackers cannot gain access to protected LuaVM instances utilized by other clients, aiming to tamper the executed scripts or monitor their results.

8.5.6 Optimizations

One of the main challenges with LuaGuardia's implementation 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 LuaVM inside the SGX enclaves, they are not optimal in terms of performance.

Initialization As described above, each client connection triggers the instantiation of a new protected LuaVM instance at the LuaGuardia server. This process begins with the initialization of a new SGX enclave and afterwards the initialization of the encapsulated LuaVM with a clean state. However, we notice that this initialization overhead can significantly affect the execution of multiple short-lived scripts that are continuously issued by a client. This happens since each consecutive execution request mandates the initialization of a new SGX enclave and LuaVM instance. To improve the performance in such scenarios, we choose to initialize the SGX enclave only once, upon the first client request. Afterwards, each subsequent initialization requires only the creation of a new clean Lua state in the protected LuaVM. As presented in our evaluation, Section 8.6.4, this design can significantly increase the overall performance as the main initialization overhead derives from the creation of the SGX enclave. Moreover, this overhead is increased as the chosen maximum enclave memory is expanded via LuaGuardia's configuration.

System Call Batching Any optimizations in the network stack, such as TCP reconfigurations or data compression, are not universally applicable as they can offer increased performance only in certain cases, or negatively affect others. Instead, we choose to explore the benefits of system call batching. The motivation behind this choice is that the system call overhead becomes significant when dealing with large I/O operations, due to the fact that SGX enclaves do not have direct access to system calls. As a result, each time the protected enclave code requires to use a system call, it has to forward the request to the untrusted part of the application, where it is served, and wait for the results to be forwarded back to the enclave. After extensive evaluation, we found that the switch between the en-

clave and its untrusted driver code is an expensive operation which negatively affects the overall performance and, in many cases, can be optimized by system call batching. By doing so, system calls that are not dependent to one another, can be buffered and served with a single, or at least much fewer, `OCALL`(s) in certain parts of the Lua script's execution.

8.6 Evaluation

In this section, we present LuaGuardia's performance and security evaluation. First, we describe possible attacks scenarios against LuaGuardia and discuss how our architecture is able to thwart them. Then, we provide a thorough performance evaluation and comparison using a series of different and diversified microbenchmarks, as well as with three popular Lua-based real-world applications, namely (i) `wkr2` [1], (ii) `pflua` [19], and (iii) a custom VPN proxy built on top of the `snabb` [24] framework.

Evaluation Setup Our main server, hosting LuaGuardia, is based on a SGX-enabled Intel Core 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 the same hardware and software specifications. We also use a system based on an Intel Core i7-6700 CPU with 16GB of RAM, acting as an HTTP server. All machines are interconnected over a 1GbE wired network. Both the client and the HTTP server host have SGX disabled in BIOS to simulate non SGX-enabled devices.

8.6.1 Security Analysis

We validate LuaGuardia's security properties by attempting different attacks and showing how our SGX-enabled runtime is able to mitigate them.

1. Data Integrity and Confidentiality It is common for attackers to exploit software vulnerabilities, either found on the target Lua script or on the LuaVM itself, to inject malicious code. Such attacks target the control flow of the executing script, oftentimes aiming to extract or tamper sensitive data. LuaGuardia encapsulates the entire LuaVM in secure enclaves — any Lua code that needs to be execute, along with the corresponding data and modules, arrive at the server securely in an encrypted format and is decrypted only when inside the VM's enclave. Since data and scripts never reside in plain-text format out of the enclave, our system is able to prevent access even to fully privileged attackers.

Similarly to incoming data, the output of the executed Lua scripts is encrypted inside the enclave and transmitted back to the client securely, in a cipher-text format. To verify this, we use `gdb` to attach the LuaGuardia process, using administrative permissions (i.e.,

root access). By attaching the process we can attempt to tamper the execution in order to dump the executing code and the data being processed. However, no such information can be extracted since the enclave's code and data are completely inaccessible from the untrusted environment, always resulting in segmentation violations.

2. Controlling the Kernel In case of a full system compromise, the execution of an unprotected LuaVM, the file system and the network interfaces can be monitored or manipulated. However, even if adversaries can monitor the network traffic between LuaGuardia and its clients, or the file system, they cannot obtain the client requests and server responses as they are encrypted throughout the entire path and are only decrypted inside the enclave. The same applies for data stored persistently in the file system as they are sealed before leaving the enclave and unsealed upon reuse. The unsealing operation takes place in the enclave and validates the persistently stored data before reuse. Moreover, even if the attackers manage 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 as the enclave's protected memory is not mappable outside of the enclave.

8.6.2 Microbenchmarks

In this section, we provide the evaluation of LuaGuardia using a series of microbenchmarks, aiming to understand and explain the variables that affect our system's performance as well as how it compares to a vanilla LuaVM. It is known that the execution performance of an SGX-enabled application is almost identical to its non-SGX variant, as long as no system calls or I/O are involved and all the required data are accessible in the enclave. However, system calls cannot be triggered from an SGX-enabled application and have to be served from the untrusted part. 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. An additional overhead, linked with the storage and manipulation of data in the enclave is the 128MB live protected memory limitation enforced by SGX version 1. Storing and accessing data that exceed the live protected memory trigger expensive page swapping that requires the encryption of the page to be swapped and the decryption of the retrieved page, each time, as we discussed in previous chapters.

Data Retrieval To understand the impact of system call serving via transferring the execution outside of the enclave, as well as the overheads introduced by triggering the protected memory page swapping, we perform the following experiments. First, we develop a

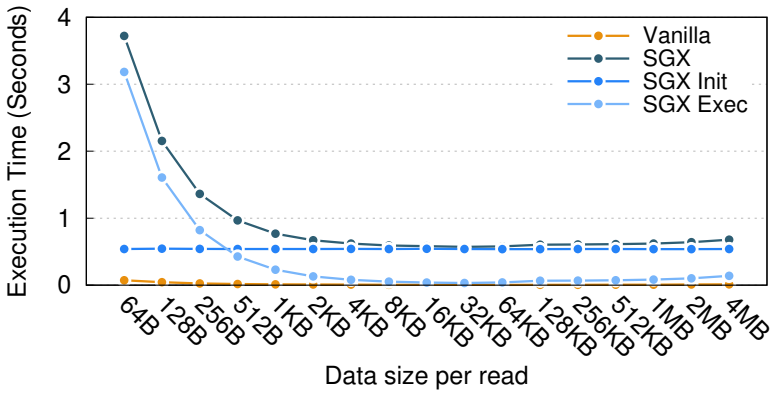


Figure 8.2: Performance comparison between the vanilla LuaVM and our SGX-protected LuaVM when reading a 32MB file with variable read buffer sizes.

simple Lua script that reads a 32MB file from the file system in chunks ranging from 64B up to 4MB. We execute the script using both the vanilla LuaVM and LuaGuardia and present the performance results in Figure 8.2, where *Vanilla* indicates the performance of the unmodified LuaVM and *SGX* the end-to-end performance of LuaGuardia. The overall execution of LuaGuardia is broken down to *SGX Init* and *SGX Exec*, indicating the time required to initialize LuaGuardia and the time required only for the Lua script 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 is transferred fewer times between the secure and unprotected spaces of LuaGuardia. Also, the encryption process for the enclave inbound data is triggered less frequently. However, we notice that increasing the read data buffer beyond 2KB has little to no effect in further increasing the performance. Furthermore, the overall end-to-end execution time of LuaGuardia is increased compared to the vanilla LuaVM due to the expensive server initialization while the required script execution time is almost identical to the stock LuaVM. This indicates that SGX does not introduce significant performance overheads when carefully designing the enclave I/O channels.

Memory Accesses The next step is to evaluate the performance overhead introduced by randomly accessing the enclave-protected memory with and without triggering page swapping. To achieve this, we design a simple Lua benchmark that performs 1 million random accesses to consecutive protected memory spaces, ranging from 1KB to 4MB. We execute the script twice, the first time performing 1-byte random writes and the second 1-byte random reads at each random access. The results of this analysis are presented in Figure 8.3. We notice that the memory access times achieved by LuaGuardia, both for read

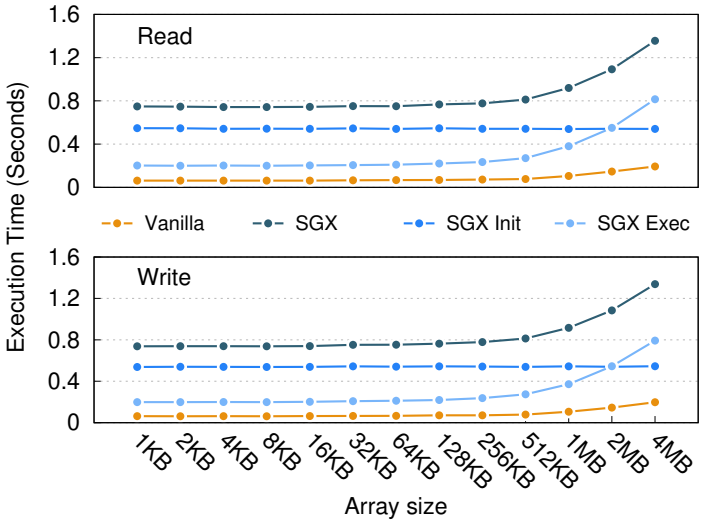


Figure 8.3: Performance comparison between the vanilla LuaVM and the SGX-enabled runtime when performing one million random accesses (read/write) to memory locations ranging between 1KB and 4MB.

and write operations, are almost identical to those achieved by the vanilla LuaVM 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 enclave memory page swapping is triggered. In such cases, we can see that page swapping affects the overall execution, enforcing a performance overhead as the amount of non-live protected memory increases.

8.6.3 Benchmark Applications

Cryptographic Benchmarks Once we understand the properties that affect LuaGuardia's performance, we proceed with our evaluation by executing 12 popular cryptographic algorithms using the vanilla LuaVM and LuaGuardia, recording only the script execution time for both 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 computations. The results are presented in Figure 8.4. The execution time is normalized to indicate the processing of 1MB input for each algorithm. Also, we present the performance overhead on top of each bar cluster, roundup to 0.5. The results indicate that the average performance overhead introduced by LuaGuardia is 23%, with SHA256 yielding the highest performance overhead, reaching 46%. This behaviour reported for SHA256 is mainly attributed to the increased memory requirements of this implementation which triggers protected memory page swapping.

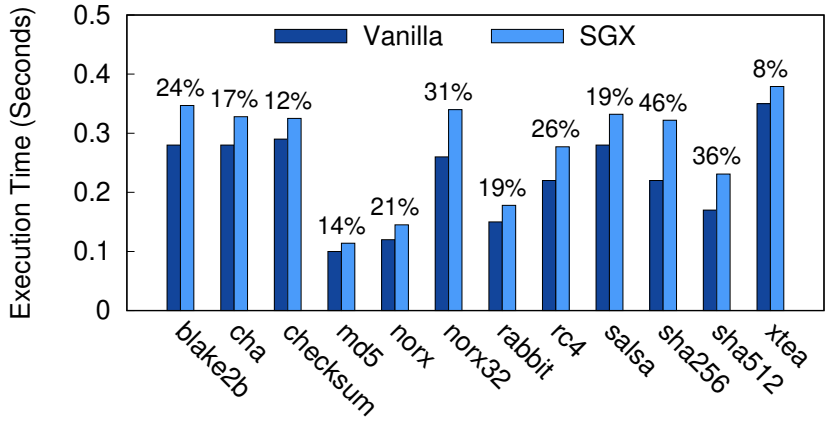


Figure 8.4: Execution time achieved by the vanilla LuaVM and LuaGuardia when executing 12 cryptographic benchmarks. The overhead introduced by LuaGuardia is indicated on top of each bar.

Benchmark Suite In this section, we evaluate and profile LuaGuardia using 12 popular benchmark applications, developed in 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. As described in Section 8.4.3, when LuaGuardia operates in *local* mode, both the target Lua scripts and their data reside on the same physical machine, in contrast to *remote* mode, where the Lua scripts and their data are securely transmitted by the client over the network.

Table 8.2 provides a short description of each benchmark 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 LuaVM when applications with different workloads are executed.

We execute each benchmark ten times, using the vanilla LuaVM and the two operation modes provided by LuaGuardia, and report the average end-to-end execution time in Figure 8.5. The values reported for LuaGuardia contain the time required to initialize the server, including the initialization of the SGX enclaves and the LuaVM, as well as the encryption and decryption operations required to secure the incoming Lua scripts, their input data, and 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 LuaVM yields significantly better results compared to LuaGuardia when benchmarks with very low computational needs are executed. However, the delta between the stock LuaVM and LuaGuardia decreases as the benchmarks become more computational intensive or demand more I/O, thus requiring more

Table 8.2: Benchmark algorithms and their operation.

Microbenchmark	Description
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	Looping recognition algorithm
nbody	n-body solar system simulation
recurs.fib	Performs recursive Fibonacci

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 the major overhead introduced by LuaGuardia mostly derives by its expensive initialization, as between executions the server is always terminated and re-initialized.

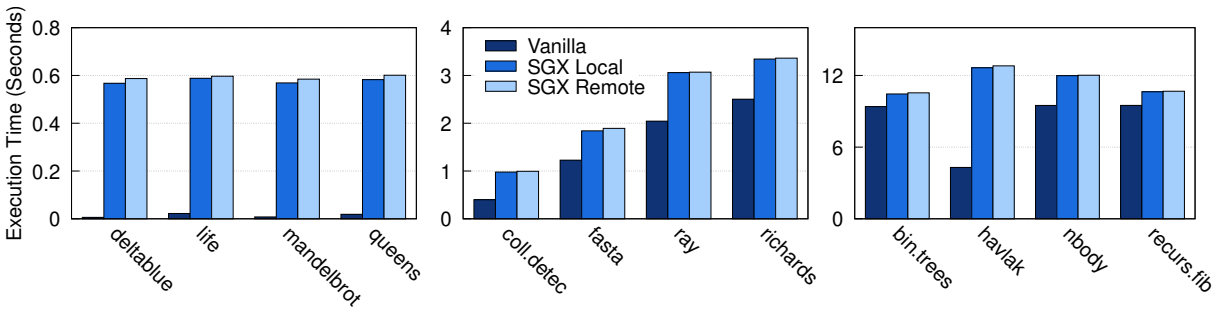


Figure 8.5: End-to-end performance comparison of the vanilla LuaVM with LuaGuardia's two operation modes when executing 12 Lua benchmarks.

8.6.4 Performance Optimizations

As described in the previous section, the initialization time of the SGX enclaves and the LuaVM introduce a significant performance overhead, especially when executing lightweight

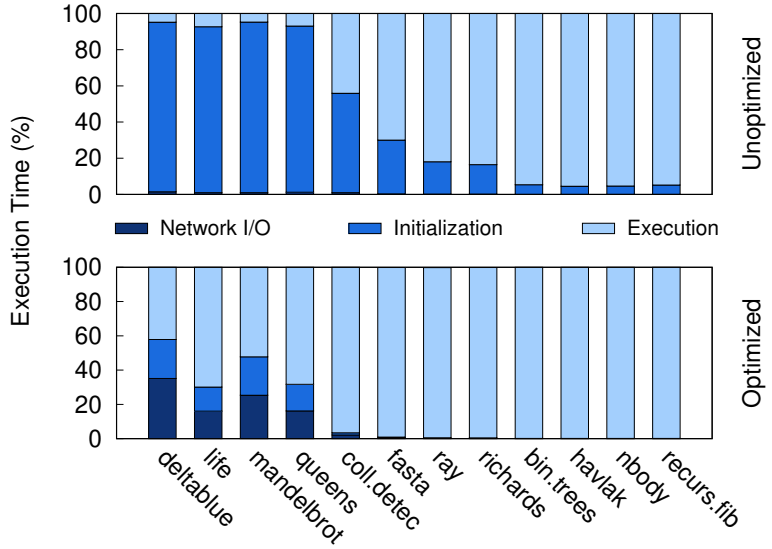


Figure 8.6: Execution time breakdown with and without initialization optimizations.

scripts that are not computation-intensive, yielding sub-second execution times. To further understand the various overheads introduced by LuaGuardia's software stack, we re-evaluate LuaGuardia with the same benchmarks, reporting the execution time breakdown into three different categories: (i) network I/O, (ii) initialization and (iii) execution. The values reported for the initialization contain the time required for initializing both the enclaves and the protected LuaVM, while the execution time includes the required cryptographic operations. As shown in Figure 8.6, 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). To overcome these start-up overheads, we re-design LuaGuardia to use pre-initialized enclaves, as described in Section 8.5.6. In particular, the enclave hosting the LuaVM, as well as its driving code, are initialized only once during bootstrap and remain active throughout the server's lifetime. Afterwards, each client request yields the initialization of the protected LuaVM, each time reusing the always-active enclave. The LuaVM is always re-initialized upon each client request, discarding the previous Lua state, to ensure the confidentiality of previous executions. As we can see in Figure 8.6, the initialization optimizations allow for faster end-to-end execution, especially for short-lived scripts.

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 ten thousand up to one million, using the vanilla LuaVM and Lu-

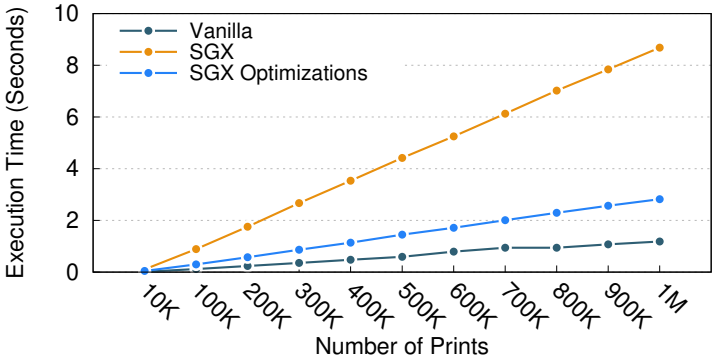


Figure 8.7: Performance comparison between the vanilla LuaVM and LuaGuardia with and without system call batching.

aGuardia with and without system call batching. 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 8.7, we notice that the overall execution performance is increased by an average of 70% for applications performing multiple system calls.

Optimization Results We re-evaluate LuaGuardia with all optimizations enabled, using the same experimental setup. Figure 8.8 compares the execution time achieved by the stock LuaVM with the unoptimized 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 LuaVM 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 LuaVM, with the only exception being `havlak` due to its high live memory requirements which constantly trigger enclave memory page swapping.

8.6.5 Real-world Application 1: wrk2

In this section we evaluate wrk2's performance when executing over LuaGuardia. wrk2 [1] 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, which at its original implementation are executed via LuaJIT. However, the scripts are also executable via the stock LuaVM, and consecutively by LuaGuardia. We choose to evaluate our system using wkr2 since it serves as a good example of networking applications that generate synthetic loads and require external services, such as an HTTP server.

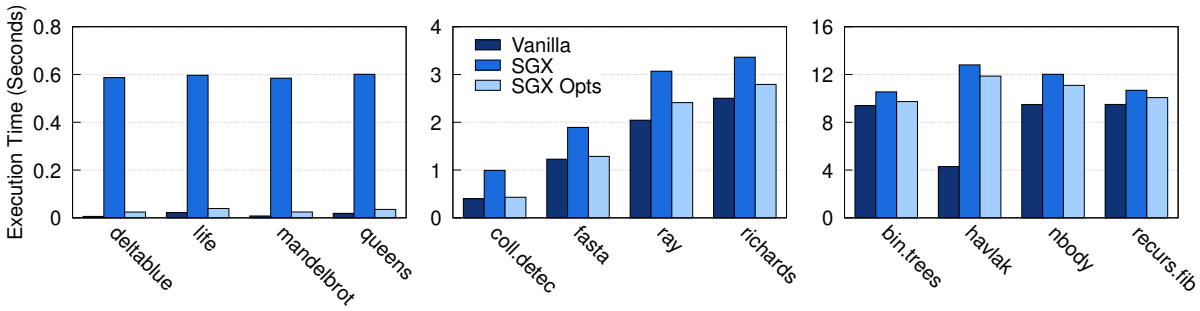


Figure 8.8: Performance comparison between the vanilla LuaVM and LuaGuardia with and without optimizations when executing 12 Lua benchmarks.

Experimental Setup Except from the two machines hosting the LuaGuardia server and client, we also setup an external HTTP server, running `darkhttpd`, which wrk2 connects to and performs 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 evaluate LuaGuardia using wrk2 by initially setting up the `darkhttpd` service on a dedicated machine. Afterwards, 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, wkr2 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 request. This means that for each packet transmitted to the HTTP server, LuaGuardia's protected LuaVM is reinitialized to execute the appropriate wrk2 script. The design followed by wrk2 serves as a good example of clients constantly requesting script executions with minimal data and computation requirements and is ideal to stress LuaGuardia's initialization optimizations.

Results The results obtained after executing the three wrk2 Lua scripts using both the stock LuaVM and LuaGuardia are presented in Figure 8.9. The X-axis indicates the executed scripts while the Y1-axis indicates the sustainable throughput in Kbps. The bar clusters follow the Y1-axis and the throughput achieved by the stock LuaVM 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

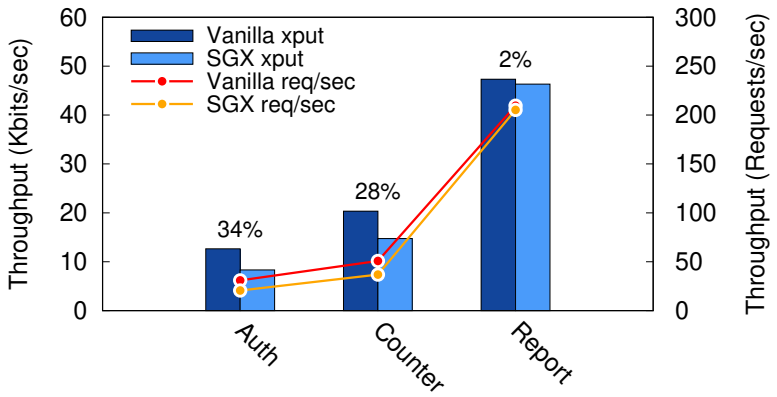


Figure 8.9: Performance comparison between the vanilla LuaVM and LuaGuardia using wrk2 with three benchmarks.

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 benchmarks. On the other hand, *Report* yields almost the same throughput since the script is only executed once and requires very minimal processing and I/O.

8.6.6 Real-world Application 2: Snabb/pflua

The second real-world application that we choose to evaluate our system with is Snabb [24]. Snabb is a virtualised Ethernet toolkit that allows the implementation of networking applications using Lua. Originally, Snabb utilizes the LuaJIT instead of the LuaVM, however, the developed scripts can be executed with the stock LuaVM 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 ~9,600K LoC, and filters incoming packets based on pflang [20], a filter that is also used by `tcpdump` [27].

Dataset To execute Snabb/pflua, we use a dataset of 12 `pcap` traces, each one containing 32,000 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. To properly evaluate LuaGuardia with these traces, we expand Snabb's synthetic traces by replaying them to increase their size to 32,000 packets. The ruleset provided to pflua contains 100 custom rules, defined in pflang, and is used to process all pcap traces. We design the ruleset in such a way that multiple rules are being triggered by each trace, making the workload uniform and comparable across all input data.

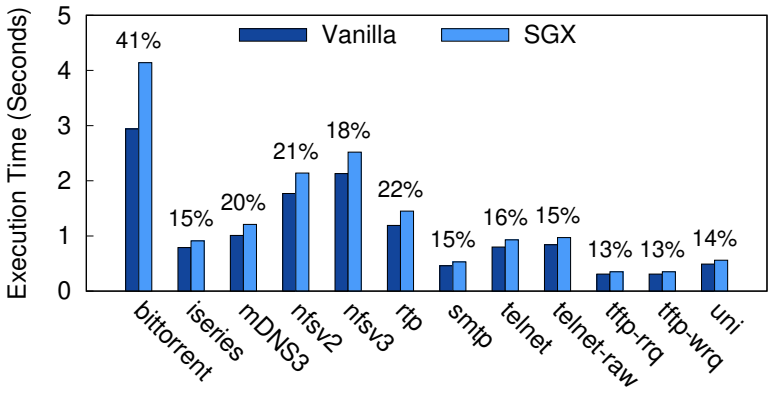


Figure 8.10: Performance comparison between the vanilla LuaVM and LuaGuardia using Snabb/pflua with 100 user-defined rules.

Evaluation Process Once all the required data and the ruleset are gathered, we proceed with the evaluation of Snabb/pflua in the following way. First, we execute Snabb to extract the packet information from each pcap trace and transform it to a pflua-compatible format. Internally, Snabb uses an 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 ruleset 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 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 LuaVM and LuaGuardia.

Results The results of this analysis are presented in Figure 8.10. 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 LuaVM 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 it is not able to hide such an overhead. In contrast, the stock LuaVM does not need to perform these costly transfer and cryptographic operations. However, we can see that our runtime is able to process the real-world traffic trace, namely `uni`, with a minimal overhead of 14%.

8.6.7 Real-world Application 3: Snabb/VPN

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 to decode pcap traces and transform them into JSON. Afterwards, the transformed packets are forwarded to our secure Lua module, executed by LuaGuardia, which is responsible for encrypting end decrypting each packet, operating as a lite VPN module.

Data We evaluate our custom module using the pcap traces described in Section 8.6.6. Since our 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 are 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 LuaVM 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/VPN 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 `SGXwritter` object. Upon initialization, we create a new `SGXreader` and `SGXwritter` 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 `SGXwritter`. Once the entire trace is processed, it is forwarded to our VPN module, executed securely by the LuaGuardia server.

Internally, the module receives the JSON-encoded packets and performs the 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/VPN using both the vanilla LuaVM as well as LuaGuardia with every optimization enabled, each time processing a different pcap trace.

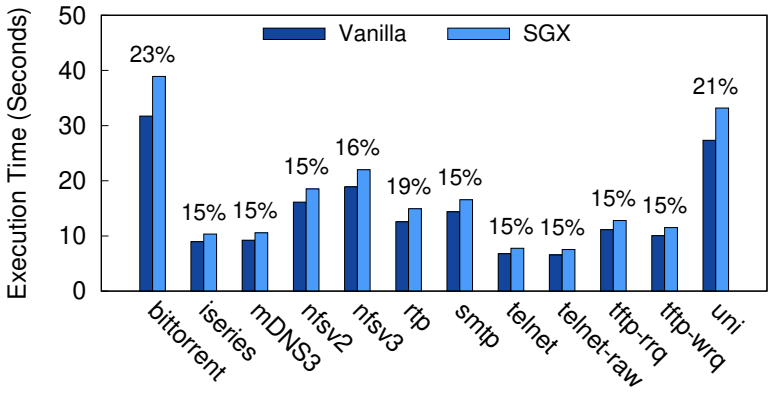


Figure 8.11: Performance comparison between the vanilla LuaVM and LuaGuardia using our custom Snabb/VPN module.

Results The evaluation results obtained after the module's execution are presented in Figure 8.11. 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 LuaVM 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 the SGX enclaves. This is more evident when comparing the overhead introduced by LuaGuardia when processing the `bittorrent` pcap trace using Snabb/pflua and Snabb/VPN. 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.

8.7 Summary

In this chapter, we presented our framework's secure Lua runtime module, a system that aims to simplify the development of confidential computing. This module addresses: (i) the lack of high-level TEE abstractions that forces the use of low-level memory- and type-unsafe abstractions, (ii) the technical issues regarding runtime extensibility, management of cryptographic operations, and restricted interfaces, and (iii) the need for manual appli-

cation partitioning, recompilation, and linking, 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 accelerate the protected code execution. Our evaluation applies LuaGuardia to a diverse set of applications, with an average overhead of 18%, the majority of which is due to I/O delays. This work's contributions are the following:

- We design and implement a lightweight code offloading system, based on user-level enclaves, that enables confidential computing with a high-level language, namely Lua, eliminating the need to learn or port code to device-specific TEEs.
- We perform several low-level optimizations, such as enclave pre-allocation and system call batching, to significantly accelerate our runtime's performance, especially when executing short-lived scripts.
- We show how our secure runtime 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 introduces an average overhead of 18% to real-world applications, compared to their unprotected counterparts.

Chapter 9
Automated Scaling of Trust-Oblivious Systems to TEEs

In this chapter, we present Atlas, our framework's sixth and final component that provides TEE-scaleout of new and legacy JavaScript applications with substantial speedups. Our system utilizes a series of techniques that allow a program oblivious to trusted execution to dynamically scale its execution over multiple TEEs, all with minimal-to-zero developer effort. As long as the developer provides the necessary hints about which parts have to be securely offloaded, averaging a couple of lines for large applications, our component automatically detects and dynamically scales the execution leveraging off-premise TEEs.

To allow delineating program fragments that are candidates for TEE scaleout, Atlas provides a domain-specific language with splittability annotations. To address correctness concerns, the system analyses the scripts to confirm that candidates for TEE scaleout do not run the risk of breaking the semantics of the original program. Atlas packs a series of automated program transformations that operate at runtime to detect load and scale program fragments over the available TEE resources. The system provides a TEE-enabled language runtime, language-aware serialization support, end-to-end encrypted communication, remote module loading and offloading, and other system functions. To develop this component, we use a JavaScript interpreter packed within user-level enclaves. Based on [50], we choose to extend, port, and embed the QuickJS [21] engine due to its superior performance compared to other embeddable JS runtimes, minimal codebase, and its ES6 [76] support. Our evaluation shows that Atlas's TEE-enabled scaleout achieves significant speedup on completely unmodified applications. Also, a series of benchmarks show attractive elasticity characteristics, with Atlas responding almost immediately to changes in load, applied to nine real-world applications.

9.1 JavaScript

JavaScript (JS) is a popular scripting and programming language that allows users to create multiple kinds of applications and multi-purpose programs. The types of applications range from small command line applications to complex web-based frameworks. JavaScript is a dynamically typed language, meaning that it is able to change the type of a variable while a script is executing. Also, any JavaScript program can terminate when an error appears at runtime. Another important feature is the module ecosystem that amplifies the modular programming characteristics via third-party modules.

From a developer's perspective, importing a module makes its functionality available to the calling code by means of binding its functionality to a name in the caller's scope. This is achieved by some form of exporting, where the module developer expresses which values should become available to the importing code.

One of the most prominent JavaScript standards is ECMAScript 6 (ES6), used among different web browsers to ensure the interoperability of web pages. ES6 includes syntactical support for classes implemented in pure JavaScript, following a similar trend to other languages with class-based features [176]. Using the keyword `class`, the developer may define a new class object, the `constructor` initializes the internal data of the class, while `extends` and `super` offer the same functionality as found in other languages.

9.2 Design

9.2.1 Atlas Components

The overall architecture of Atlas is presented in Figure 9.1. The system consists of two entities: (i) the *client*, which can be any device able to execute code, and (ii) the *SGX workers* that reside in the cloud. The client holds all the logic for wrapping the function call(s) with all necessary data and metadata as well as the scheduler's logic. Each Atlas computing node residing in the cloud is equipped with the QuickJS JavaScript engine, encapsulated within SGX enclaves, thus preventing kind of malicious operation on the served data.

9.2.2 Atlas Execution Flow

In this section, we present a step-by-step execution flow on the Atlas environment. The two principal components in the setup are: (i) the client, which is the one initiating and requesting task distribution, and (ii) the trusted SGX-enabled Atlas worker nodes, residing in a remote infrastructure.

Let's assume that every available Atlas node is known and available via a public configuration file, including all the required metadata to establish a new connection. First, the client parses the configuration and initiates a connection request to each worker it wishes

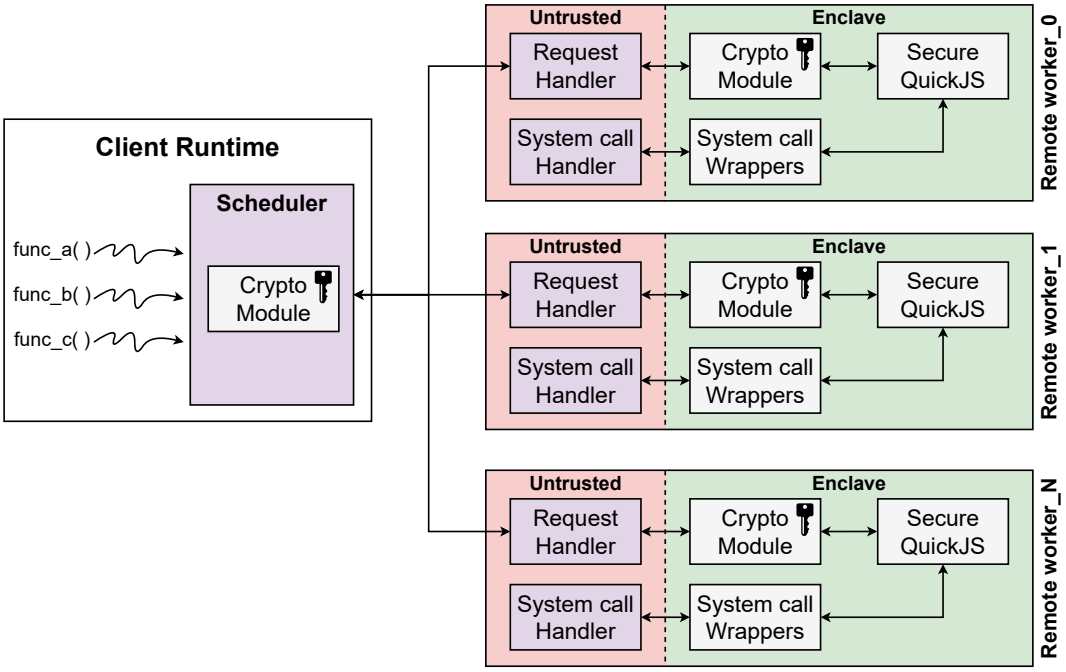


Figure 9.1: Atlas architecture and execution life cycle overview.

to utilize and each worker accepts the connection and marks the client as a source for incoming requests. Once the connections are established, the client performs a handshake with each worker using asymmetric keys, as described in previous chapters. After this point, the client and the workers generate unique symmetric cipher keys which are always stored in the SGX enclaves at the worker's side. Once the secure channel is established, the work distribution can begin.

The next step requires the developer to define which functions will be scaled-out to the remote Atlas workers by simply wrapping them with the `atlas_wrap()` function. At this point, the executable script is forwarded to the client's modified QuickJS engine and it is analyzed. For each wrapped function encountered by the parser, the Atlas task scheduler picks the next available worker, crafts a new message containing the function code, arguments and other metadata required for the remote execution. The message is then serialized, encrypted, and forwarded to the respective Atlas worker.

When a worker receives the offloading request, it forwards the encrypted blob to its trusted counterpart, residing in the enclave, where the decryption takes place. The plain text then is de-serialized and evaluated inside the worker's protected QuickJS runtime. Once the execution is completed, the results are serialized and encrypted within the enclave. The encrypted blob containing the results is then returned to the worker's untrusted part, out of the enclave, and is forwarded back to the client. Finally, the client gathers all the results from each worker and proceeds with its normal execution.

9.2.3 Atlas Workers

As mentioned in the previous sections, Atlas workers play the most vital role in the Atlas ecosystem. They are the entity responsible for handling incoming requests, de-serializing execution requests, and most importantly preserving and ensuring the integrity of the code execution and client data. A trusted worker contains several discrete sub-modules that reside inside the SGX enclaves. To establish an end-to-end communication channel with the client, Atlas offers a cryptographic engine responsible for handling messages exchanged between the client and the remote workers. The cryptographic engine along with any cryptographic keys used for the encrypted communication and the handshake only reside inside the enclave, thus, no code or data are ever exposed to the untrusted DRAM.

Another critical Atlas worker component is the serialization engine responsible to marshal the messages before and after any cryptographic operations are performed. This engine can serialize complex data structures, objects, basic data types, and even pure JavaScript functions, assuming that the code can be deemed safe. Moreover, since the code executing inside the Atlas worker's enclave is developed in pure JavaScript, direct system calls and other I/O related functions have to be forwarded out of the enclave, as the SGX execution model dictates.

9.2.4 Scaling Out

The Atlas scheduler is our system's component responsible for client side message handling, worker allocation, and task distribution. More specifically, Atlas supports two configurations for remote scheduling: (i) static allocation, where the number of workers and servers are pre-defined in user-provided configuration files and are simultaneously allocated during the initialization of the client end-point, and, (ii) dynamic allocation, where the number of workers increases proportionally to the average message latency. In the latter, the user has the option to provide a threshold that will trigger the dynamic node allocation or use the default values, provided by Atlas's standard configuration. The requests are served to the workers based on their availability in a simple round-robin fashion.

9.3 Implementation

In this section, we present Atlas's implementation details, the challenges encountered during the development and the enclave-porting process, as well as the optimizations we applied on the system to boost its overall performance.

9.3.1 Porting QuickJS

Atlas is based on the QuickJS JavaScript engine [21] which is developed in pure ANSI C, making it a great candidate for our needs. Compiling the original QuickJS codebase produces two distinct binaries, the first one being a high-level interpreter that parses and executes ES6 compliant JavaScript code, named `qjs`, while the second binary is capable of compiling and executing JavaScript code into native code, called `qjsc`. We choose to utilize the former since native modules developed in pure C, combined with the QuickJS low level API, are not supported by our implementation as we describe in Section 9.5.1.

The official QuickJS codebase consists of ~90K LoC and Atlas utilizes ~80K LoC from the original sources. As described in previous chapters, the Intel SGX1 execution model limits the number of functions available to the enclaves. System calls for example cannot be served within the enclave and thus several changes have to be performed to the runtime's components, either by wrapping and proxying the unsupported functions or excluding them entirely but without breaking legacy compatibility of pure JavaScript code. Some JavaScript components are completely scrapped off the original codebase to provide enclave support without exposing the system to external threats. For example, SGX version 1 does not support the creation and allocation of threads within a single enclave. Additionally, code inside the trusted Atlas workers is executed sequentially. For these reasons we choose to remove the `ATOMICS` and `WORKERS` module from our runtime.

Porting the QuickJS runtime to the Atlas worker's enclaves is not a straightforward task. First, due to the lack of system call support in the enclave, several functions related to the execution environment have to be modified. As a result, *signals* are completely removed from the codebase. To avoid breaking normal normal code execution, we have to manually provide trusted bi-directional bridges from the enclave to the untrusted part of Atlas and vice versa to support the system calls required for the runtime's execution.

Apart from the trusted interpreter, we also have to provide the necessary functionality to the untrusted part of the modified QuickJS engine to transform it to a functional worker. The main two components that have to be added is the `Request Handler` and the `Function Handler`. The former is responsible to manage client requests, forward the results, and interface with the enclave appropriately to help establishing the communication channel. The latter is responsible for serving function execution requests initiated by the enclave, in cases where they cannot be served within the enclave (e.g., system calls).

9.3.2 Bootstrapping

Enclave creation and initialization are costly operations. To ameliorate these unwanted costs that impact the workers' performance and slow down the overall execution, we choose to perform the following optimizations: (i) initialize the enclave and load the required libraries during the enclave creation time, and (ii) re-use the same enclave with the pre-

loaded libraries for each new client offloading request, after having reset the enclave's state. In this way, for each execution request, we provide an enclave with all the required code, common libraries and modules, since the initialization occurs only once. After this point, for each new client request, the only thing that has to be transferred to each Atlas worker is the encrypted blob containing the offloaded code, arguments, data, and custom modules. Once a previously unloaded module or library is loaded for a specific client, the client's trusted module database is updated so new remote execution requests from the same client can utilize the available custom modules.

9.3.3 Distribution Steps

For a client to successfully offload and scaleout tasks to the remote Atlas workers, a set of steps has to be followed as described bellow.

1. Serialization

Prior to offloading task execution requests to an Atlas worker, we have to first serialize each client message. A typical client request contains information and metadata related to the required libraries, the target function that has to be executed, its respective arguments, that may either be simple data types or more complex data structures (e.g., graphs, hash-maps, arrays, lists, etc.) and the worker's unique identifier. To do so, we have extended the functionality of the built-in JavaScript functions offered by the vanilla QuickJS engine, such as `JSON_parse()` and `JSON_stringify()`, to support the serialization of the client's message. Our custom parsing process can be invoked using the `atlas.parse()` and `atlas.stringify()` functions. The client can call `atlas.stringify()` to serialize its data blob before sending it to the remote worker. Similarly, each Atlas remote worker has to invoke `atlas.parse()` to de-serialize the data. The reverse procedure is performed when the worker sends the results back to the client.

2. Encryption

Since the main goal of this component is deploying and distributing tasks on remote workers, we assume that malicious parties may have full control of the network channel and can monitor Atlas's traffic and connections. To overcome this issue, we add an extra encryption layer to the serialized data before sending them on the wire. Our main goal is to offer fast encryption/decryption operations while achieving the best security to performance ratio. For this reason, we extend our custom QuickJS implementation in the client-side to natively provide encryption, decryption, and key generation functionalities in pure C. This is achieved by utilizing the `tweetnacl` [51] toolkit that provides streaming encryption and decryption, implemented in native C. The Atlas clients perform the re-

quired cryptographic operations transparently, via their modified QuickJS engine. At the Atlas worker side, the cryptographic operations are strictly performed in the SGX enclaves to prevent code, data and metadata from being exposed in the untrusted DRAM or storage in plain-text format.

3. Networking

The final piece in the remote execution process is secure networking which is split to two separate steps: (i) secure end-to-end communication establishment and, (ii) data exchange. To achieve secure communication, we initiate simple TCP connections from the client(s) to each one of the Atlas workers that we need to utilize by modifying the vanilla QuickJS implementation. This socket connection is transparently performed when a client performs a task offloading request. We opt for such a simple and lightweight approach since the only functionality required in this case is managing a standard socket, as the cryptographic operations are already handled in the previous step.

Once all the steps mentioned in this section are completed, the client can start to asynchronously offload execution requests to each worker based on a simple round-robin algorithm. The client spawns a group of local workers, each one assigned with a remote Atlas worker, based on the configuration. This option reduces the need for synchronization primitives, since two client workers cannot access the same remote Atlas worker at once. Furthermore, both local client workers as well as the remote Atlas workers are not required to generate a unique session key for each of the possible connections.

To provide optional, on-demand, secure communication between the client(s) and the remote worker(s), we provide a simple JavaScript API that transparently binds the network operations with the cryptographic operations discussed above. This API contains four functions, namely `atlas.connect()`, `atlas.close()`, `atlas.send()`, and `atlas.recv()`. Using these functions, we can perform cryptographic handshakes with all the remote workers in our setup, and consequently exchange cryptographic keys to protect the data.

9.3.4 External Modules

By design, JavaScript is a programming language that offers dynamic module capabilities in two ways and has built-in support for two types of libraries: (i) libraries that are implemented entirely in pure ES6 compliant code, and (ii) libraries that are developed as native modules, built in C/C++, that may be registered using the native QuickJS C API. To support pure JavaScript module loading, there are two main functionalities that have to be supported inside the enclave. The first is the correct operation of the `FILE` structure, required to handle module files and I/O. Second, all the functions invoked during the bootstrapping phase of the QuickJS engine such as `fopen`, `fread` and `fwrite` have to be available. These functions are required to fetch and load QuickJS compliant encrypted modules from

the untrusted file system and perform the decryption and necessary content verification in the trusted part of the Atlas runtime for integrity purposes. With these functionalities in place, our secure runtime can load pure JavaScript modules and start executing them. This specific design prevents attackers from modifying or replacing modules found in the host system's memory/file system or modules received over the network, as part of a remote execution request. The checksums required for module validation can either be pre-stored in Atlas's worker enclaves, such as checksums for standard libraries, received in encrypted format during the data transferring phase, or be part of the encrypted blob, if SGX Data Sealing is used. With this mechanism in place, any `import` or `require` snippets encountered in a JS script during the execution within the Atlas enclave assume that the target source code exists in the untrusted file system in an encrypted format and can thus be loaded. This intermediate loading layer is transparent to the developer and offers the same functionality as the vanilla QuickJS module loading process. To further optimize this process, Atlas's standard configuration pre-loads all the standard libraries and a set of popular JavaScript modules to speed up the script execution.

On the other hand, native C/C++ libraries, are compiled as shared objects using QuickJS's C API and may only be loaded by Atlas's untrusted part, residing out of the enclave, using the `dl*` family of functions. However, this functionality prevents us from providing trusted dynamic C/C++ library support for the following reasons. First, since shared objects cannot be loaded in Atlas's enclave (`dlopen` is a system call whose functionality cannot be wrapped and proxied), their integrity cannot be verified by the system. Second, the functions loaded in Atlas's untrusted part may only interface with the enclave code via a predefined, before compilation, proxy layer called Enclave Definition Language (EDL) file. Thus, each function's prototype is not known a priori and this proxy layer cannot be constructed. Additionally, since the shared objects are stored in the file system, and the `dl*` family of functions is not available in the enclave, module verification cannot be performed enabling a malicious entity to tamper the native module.

9.3.5 System Call Handling

The most common issue with many TEE technologies, including Intel SGX, is the lack of system call support. This is expected since only the underlying hardware and the SDK provided by the vendor are considered as trusted components, thus system calls, peripherals and even the OS kernel are considered untrusted and excluded from the TCB. So, directly performing calls to the untrusted kernel, such as I/O and network operations, is not always considered sound since a malicious user can intercept the calls and control/monitor the context and the data of each call. When utilizing SGX, such calls have to be offloaded to the untrusted part of the application, using proxies defined in the EDL file, where they are handled, and the results are forwarded back to the enclave. Meanwhile, the enclave

has no means of validating the results unless they are generated in an attestable way.

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

The most common approach to handle this issue is to implement custom system call wrappers for each required system call. However, modern operating systems provide hundreds of system calls, but the majority of them are not required by the QuickJS runtime or the executing scripts. 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 reason, we decide to provide interfaces only for the bare minimum of the system calls required to support the normal operation of the interpreter inside the enclave and minimize the potential attack surface without having to keep proxying functions calls to the untrusted part of the Atlas runtime.

By analysing the original source code, we notice that the most commonly used functions resulting to system calls are those associated with data I/O such as `fopen`, `fwrite`, `fread`, etc. To provide system call support to the JavaScript engine residing within the SGX enclaves, we have to develop custom functions, defined in the EDL file and implemented in the non-enclave part of the Atlas worker, performing OCALLs for each pending request. Once such a request is handled to the OCALL interface, and since the untrusted part of the application has full access to the entire system, it will be served and the results will be returned to the enclave after performing the typical SGX-enforced checks on the data. However, the validity of the data transferred from the untrusted part within the enclave has to be explicitly checked for integrity with mechanisms that have to be implemented by the application developer, in cases were the returned data are critical and can lead to landing an attack. In the Atlas model, extending SGX's model, the only entity assumed trusted is the enclave, the SGX driver, and the SGX services and SDK, whereas the JS scripts, modules and input data required for the execution must be provided with each client request in encrypted and attestable format. Thus, we prohibit read and write operations to arbitrary file system locations at the worker's side. To help minimize the I/O constraints as much as possible, we provide optional file offloading and handling in predefined Atlas worker file system locations in an encrypted manner, using SGX's sealing and unsealing functionality, enabling secure and persistent storage in the untrusted file system.

9.4 Evaluation

In this section, we evaluate Atlas's security and performance characteristics. First, we describe the methodology used for the evaluation by presenting the experimental setup and the real-world applications we use to put Atlas under test. Then, we discuss our system's dynamic response and scalability characteristics. Finally, we conclude with a series of benchmarks and microbenchmarks that provide insight into our system's properties.

9.4.1 Evaluation Setup

We evaluate Atlas in two distinct test beds, as follows:

Large Multiprocessor This test bed is based on an Intel Xeon E7-8830 processor, clocked at 2.13GHz, providing 8 physical cores. The system is equipped with 512GB of memory and runs on Debian 4.19.160-2 with Linux kernel version 4.19.0-13. This setup is used as a local multiprocessor, using the official Intel SGX SDK and PSW.

Small Distributed Cluster The remote distributed cluster used for the evaluation of Atlas is based on the Azure cloud services and it contains several DCsv2 machines which offer native Intel SGX hardware support. The nodes are equipped with Intel Xeon E-2288G processors and we utilize one virtual CPU and 4GB of memory per instance. The installed Operating System is Ubuntu Server 18.04 LTS - Gen 2 with Linux kernel version 5.4.0-147.

9.4.2 Applications

To evaluate Atlas's performance capabilities, we use the following real-world applications:

Contact Discovery This application is a custom lite implementation of Signal's [23] contact discovery service in JavaScript. It uses an oblivious hash table containing all submitted contacts. The application collects a batch of contact discovery requests and builds a hash table containing all submitted contacts. Then, it iterates over the list of all registered users and, for each registered user, indexes the hash table containing the batch of client contacts, performing a constant-time comparison against every contact in that line. In this application, we send a discovery message per request that contains 100 user contacts with some additional metadata. As requests arrive to each node, the list of all the registered users is iterated and the node performs a comparison against every received contact.

Secure Hashing This cryptographic benchmark executes the SHA-512 secure hash algorithm, implemented in pure JavaScript. Upon receiving the input, the application responds with its hash value. The client generates a buffer of fixed length (500KB).

Natural Language Processing This real-world application performs natural language processing using the Compromise [16] Natural Language Processing library which has a codebase of 375KB. On each offloading request, we send chunks of document data (sized at 1.629KB). The application first finds all the *verbs*, *nouns*, *adjectives* and *adverbs* in the input. Then, it transforms all the *verbs* to past and present tense. Finally, the application transforms all the *nouns* to singular and plural form.

Unweighted Shortest-Path In this application, each worker executes Dijkstra's algorithm to find the shortest path from the source to the destination. The client sends the size of the graph to be generated. The application works in three phases: (i) generation of the distinct vertices based on the user input, (ii) construction of the graph that connects all the vertices through edges, (iii) execution of the actual shortest distance calculation algorithm.

K-means Clustering K-means Clustering is an unsupervised machine learning algorithm, meaning that it makes inferences from the given dataset using only input vectors without referring to known or labelled outcomes. The main goal of the algorithm is to group similar data points in order to discover underlying patterns. Upon receiving the user data, the application generates an array of 2500 data-points to cluster, in a *(x, y, z)* format, and sets the number of iterations to max. In a similar way, an array of `centers` is generated and passed as an argument for the initialization. Finally, the algorithm returns a cluster of identifiers for each data dot and centroids containing the array of indexes for the clusters, the array of the resulting centroids and the number of iterations it took to converge.

Decision-Tree Learning Decision-Tree Learning is a supervised machine learning algorithm, meaning that the user has to specify what the input represents and what the corresponding output is in the training data, where the data continuously keep splitting based on a certain parameter. The algorithm uses a decision tree that consists of two primary entities: (i) *decision nodes* and (ii) *leaves*. Decision nodes is the point where the data are split and may contain filters whereas leaves are the final outcome that may be either a match or not. The algorithm first generates two arrays (as set and values) with 2000 entries and then trains the decision tree using regression as the gain function, to obtain the best split. Finally, it predicts the values given the set array as input matrix.

Lexicographic Sorting This real-world application executes the Gnome sorting algorithm. Upon message arrival, the application generates an array of 2500 random numerical entries to be sorted, performs a series of swap operations and then responds to the user with an array containing every integer in descending order.

Term Edit Distance The Term Edit Distance application tries to quantify how dissimilar two character sequences are by finding the minimum number of steps required to transform the one into the other. Upon receiving the user input, the application generates two strings of 2000 bytes and by performing a series of subtractions and `Min` calculations, returns the minimum distance between the two strings.

Password Manager This application implements a simple password manager that provides four basic operations: (i) dynamic password database creation across Atlas's nodes, (ii) password and metadata retrieval, (iii) database entry modification and (iv) entry deletion. We use this application to evaluate Atlas's performance by executing thousands of the supported commands. The client packs a message with the size of the database to be generated (10K). Then 1 million operations are performed on each node. The application iterates all the available database entries and then compares each entry with the operation ID. If the comparison is correct, the fields of the found entry are modified and then the application handles the next operation.

More Benchmarks We further explore Atlas's performance characteristics using a set of sorting algorithms and a benchmark suite. The sorting algorithms set contains seven benchmarks, namely Merge-, Heap-, Bubble-, Cocktail-, Gnome-, Insertion- and Radix-sort. Finally, the benchmark suite contains seven algorithms found in real-world JavaScript applications, with different memory requirements.

9.4.3 Dynamic Response

In this section, we evaluate how Atlas responds to load spikes. To do so, we execute the *Contact Discovery* application with the following setup. We allocate four servers, hosted in the Azure cloud, providing SGX capabilities in hardware mode. Also, we allocate a local QuickJS process responsible to execute the application on the large multiprocessor. Then, we execute the contact discovery application on both systems and compare the results.

Bootstrap To bootstrap our application, we generate a random list of 160,000 entries which are split to four sets of 40,000 entries. Each set is transmitted to the four servers residing in the Azure cloud. The client issues a `bootstrap` request to each server instructing them to generate their private contact database. Once this process is completed, a `done` reply is sent back to the client. Then, the client generates locally a record of 100 contacts along with their metadata. Finally, the client issues the contact discovery requests, based on this set, at each worker. This process is performed 10 times to warm up the system.

Request Offloading Once the bootstrapping phase is completed, the application enters the main execution phase. At this point the worker queues are empty and the client starts to issue contact discovery requests, based on our custom interval algorithm. The client serializes and encrypts the data, and forwards them to the workers using the Atlas scheduler.

Intervals Algorithm To generate load spikes, we implement an interval-based algorithm which the client uses to issue the contact discovery requests. The algorithm issues a variable number of requests with six distinct intervals, each starting at the reference times T0 to T5, as follows. The first interval, T0, is 800ms at which the client issues 11 requests. Then, at T1, the interval is set at 700ms and the load at 15 requests. At T2, the interval is set at 400ms and the client offloads 30 requests. Then, at T3, the interval is decreased to 200ms, so that the requests are rapidly issued, and the client offloads 25 requests. At T4, the algorithm tries to reduce the load by increasing the interval to 500ms and the load is set to 20 requests. Finally, at T5, the interval is set at 900ms and the client offloads 20 requests. Each request contains 100 contacts to be discovered. The benchmark is completed when the entire request load is served and the client issues a session termination message. The goal of this algorithm is to emulate load spikes that appear in real workloads and enable us to assess our system's dynamic response capabilities.

Contact Discovery Process Each server performs the execution sequentially and only a single request message is received at a time. Upon a contact discovery request, each server parses the input, de-serializes the included contacts and their metadata, and then starts comparing each client contact with the existing contact database. For every accessed contact in the database, it writes a value in the respective entry in a way that a malicious party cannot track the memory access patterns. When the process is finished, and all the memory entries are touched, the attacker cannot identify which contact corresponds to which memory row. Finally, each worker responds with the serialized and encrypted results.

Evaluation Results We execute the contact discovery application as described above and report the results in Figure 9.2. The first sub-figure presents the ingress throughput – the rate at which the client issues contact discovery requests to the workers. The Y-axis indicates the issued requests per second while the X-axis indicates the wall clock time in seconds which is common for all three sub-plots in this figure. The labels E0 to E5 mark the interval change events, happening at the reference times T0 to T5, as described above. The peaks and troughs reported after the occurrence of each event are caused by shifting the interval value and number of issued requests according to the algorithm.

The second sub-figure presents our system's response latency when executed on the multiprocessor. The X-axis remains the same while the Y-axis indicates how much time is required by Atlas's workers to serve each contact discovery request. The points marked

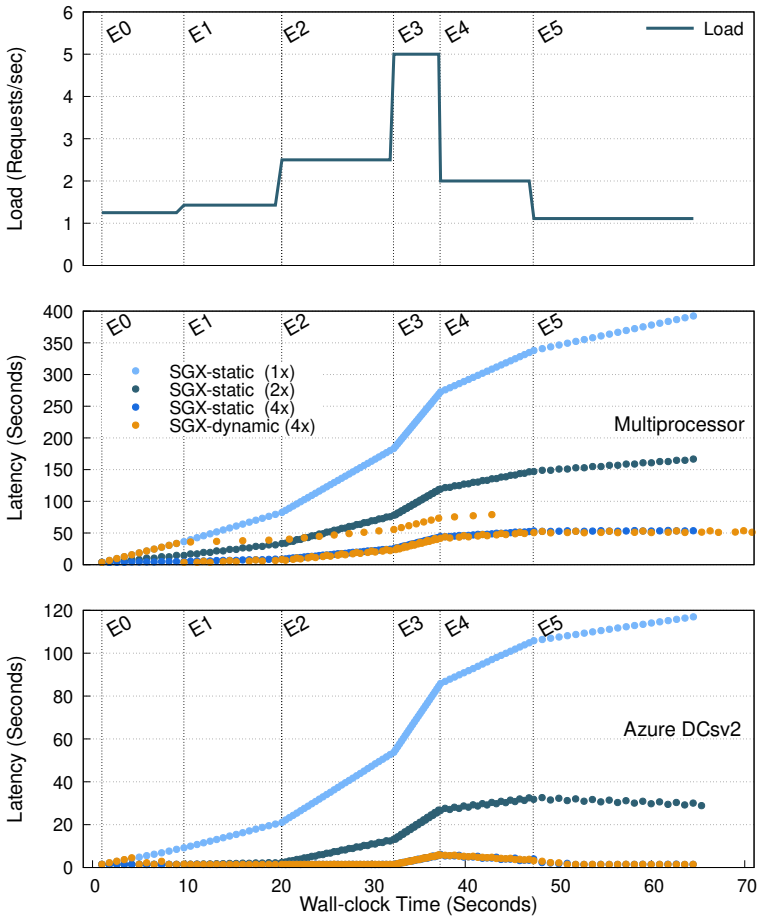


Figure 9.2: Dynamic response capabilities when executing the contact discovery application on the multiprocessor and the cloud with variable load spikes.

as `SGX-static-*` indicate the number of pre-assigned nodes used. The orange points, labeled `SGX-dynamic`, report the response latency when taking advantage of the dynamic scheduler that detects incoming load and allocates more worker nodes, in this case up to four. We notice high latency during the first events despite the dynamic mode. This happens because before dynamic scheduling is triggered, the requests are stacking up on the first remote worker. Then, when the system detects heavy load, it starts allocating more remote workers with empty work queues but the first worker has already been assigned with a big load to be processed. This is the reason why some orange data points are higher even after the dynamic scheduling is enabled. Once the extra load is processed (E4 to E5), the first worker catches up with the rest and the overall response latency decreases.

The third sub-figure reports Atlas's response latency when executed on the Azure cloud instances, leveraging SGX enclaves in hardware mode, and provides a solid picture of our

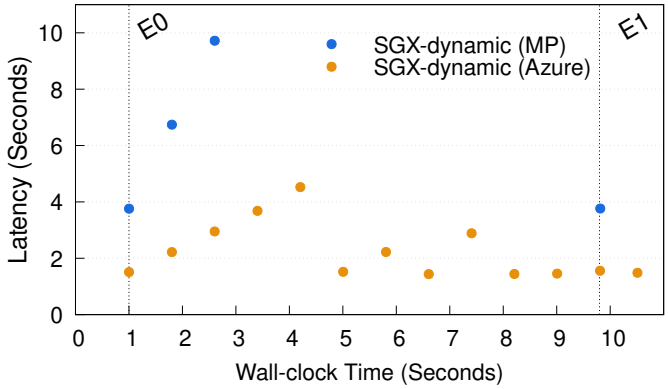


Figure 9.3: Dynamic response capabilities during the first event (E0) when executing the contact discovery application on both setups.

system's real capabilities. The first thing we notice is that the execution time is much lower since the cloud is equipped with newer and more powerful hardware while each worker resides in a separate instance. For this reason as well, Atlas is able to detect the latency changes faster and the dynamic scheduling is enabled during the half of the first interval. This is also clearly shown in Figure 9.3 which focuses on the first event (E0). This means that fewer requests are assigned to the first remote worker as more workers are spawned during the first four seconds.

To understand if the achieved performance gain is consistent among other use cases, we perform the same evaluation using eight more real-world applications, as described in Section 9.4.2. We summarise our findings in Figure 9.4, reporting the sustained latency during each event (E0 to E5) for every application, using remote workers on Azure cloud instances. We notice that the addition of multiple workers provides a significant improvement to our system's ability to handle load spikes. On E3 event, when the client starts offloading execution requests more aggressively, the overall response latency starts to increase rapidly. On this event, a single worker is not able to handle the incoming requests, while using four workers masks the load spike almost completely.

Also, we notice that in cases where the time required to serve a single request is significantly higher than the request interval, the system requires more time to recover. Unweighted Shortest-Path is the most prominent example due to its continuous array generation and costly graph calculations, needing several seconds to process a single request. For this reason as well, we observe some high latency results spanning beyond E4 when dynamic scheduling is utilized, as multiple events are still pending on the first worker's queues. This behaviour is not observed when all four workers are statically pre-allocated, indicating the need for a more aggressive scalability configuration. Finally, on E4-E5, the

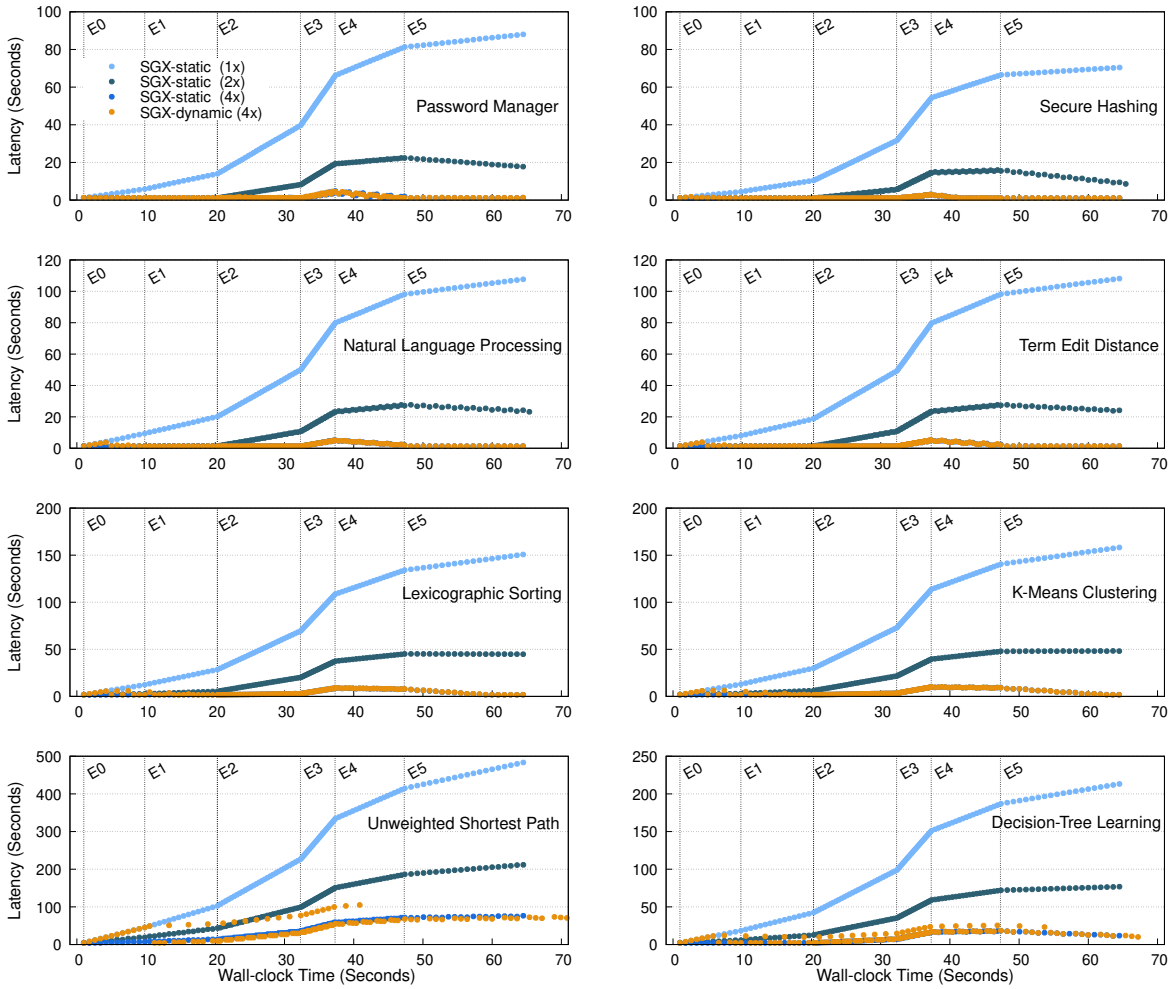


Figure 9.4: Dynamic response capabilities when executing the real-world applications using Azure cloud instances with variable load spikes.

offloading frequency starts to decline. However, when using a single worker, the response latency still increases due to the pending requests. At the E4 and E5 events, the system is able to recover from the load spike, in most cases, when using multiple remote workers.

9.4.4 Scalability Characteristics

We proceed our evaluation aiming to identify our system's scalability characteristics when executed on distributed clusters, allowing for multiple worker instances. To do so, we execute the real-world applications, described in Section 9.4.2, using Atlas workers residing in the Azure cloud. The evaluation process is performed as follows. We begin the assessment by executing each application 10 times, using a single worker and a significant workload

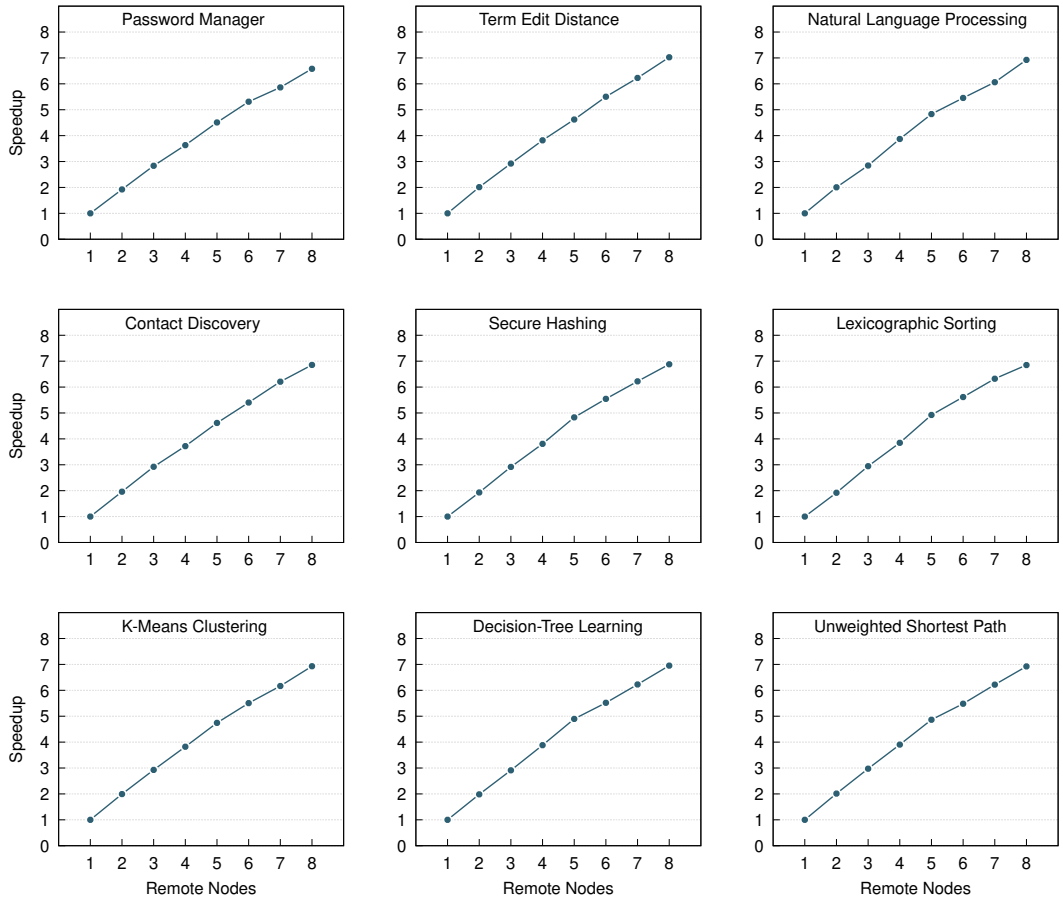


Figure 9.5: Atlas speedup gain on Azure cloud using up to eight worker instances when executing real-world applications.

that can later be distributed to multiple workers. Then, we calculate average execution time required to complete each application and set this value as its performance baseline. Next, we repeat this process 7 more times using the same workloads, each time increasing the available Azure instances by one, thus adding an extra worker. Once a set of 10 executions with the same configuration is completed, we calculate the average execution time and report the achievable speedup based on the application's the sequential execution (i.e., using only one Atlas remote worker).

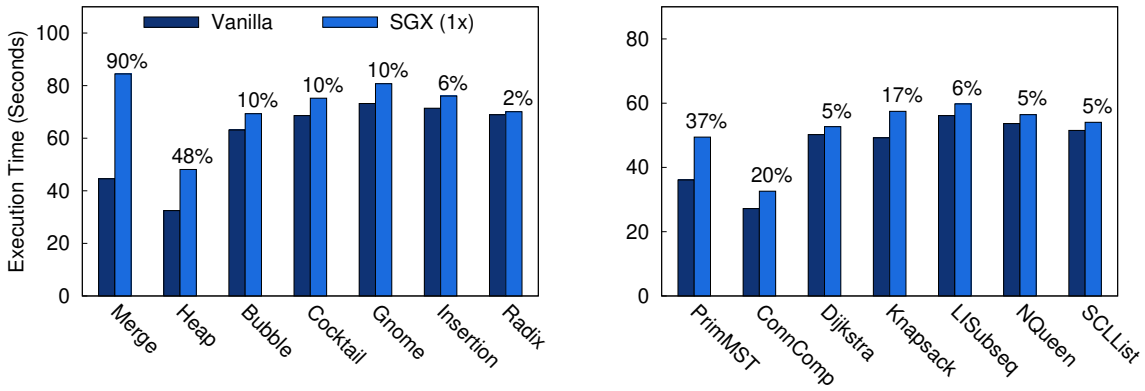
The results of this analysis are presented in Figure 9.5. As we can see in the figure, increasing the number of available workers provides a noticeable performance benefit to Atlas. We notice that for most applications, the gained speedup is linear to the number of utilized Atlas remote workers when adding up to five Azure instances. On average, the speedup gain when utilizing 8 remote workers is $\sim 6.9x$.

9.4.5 Benchmarks

In this section, we evaluate Atlas with a series of benchmarks containing algorithms with different properties and summarize the reported results. With this evaluation we aim to assess Atlas's performance when executing various types of algorithms and identify the performance overhead that SGX enclaves introduce.

We execute the sorting algorithms and the algorithms found in the benchmark suite, first using the vanilla QuickJS engine and then using Atlas with a single worker node. We choose appropriate workloads so that both the vanilla and the SGX-enabled engine require several seconds to complete the execution, limiting the I/O and bootstrap costs. The results of this analysis are presented in Figure 9.6. The Y-axis indicates the overall execution time while the introduced overhead is presented on top of each bar cluster.

The reported results show that Atlas with a single node introduces an average performance overhead of ~25% when executing the sorting algorithms, while four out of the seven are executed with a performance penalty of no more than 10%. The highest overhead is reported for Merge-sort, at 90%, mainly due to the extensive memory requirements of this particular implementation, triggering expensive encrypted memory page swapping. Similar results are also reported for the execution of the algorithms found in the benchmark suite. Four out of the seven benchmarks are executed by Atlas with up to 6% performance overhead while the average performance penalty is ~13.6%. In a similar fashion, algorithms generating multiple memory allocation requests tend to execute with higher performance overhead.



(a) Sorting Algorithms

(b) Benchmark Suite

Figure 9.6: Performance comparison between the vanilla QuickJS implementation and Atlas with a single worker node when executing various benchmarks.

9.4.6 Microbenchmarks

In this section, we present the evaluation of Atlas with a series of microbenchmarks, reporting how Intel SGX enclaves affect our system's performance.

Data Retrieval Performance

To understand the impact of retrieving data from the untrusted environment in the secure enclave, we develop a simple JavaScript benchmark that reads a 20MB file from the file system in chunks ranging from 64B up to 4MB. Then, we execute the script using both the vanilla QuickJS engine and Atlas with a single node and report the results in Figure 9.7. We mark the execution time required by the vanilla QuickJS interpreter as `Vanilla` and the end-to-end time required by Atlas's SGX enclave as `SGX`. Furthermore, we present the time required for booting our system from scratch as `SGX Init` while `SGX Exec` marks the actual execution time of the JS script inside the enclave. We notice that increasing the read data buffer, significantly reduces the script's execution time as fewer system calls are involved and the execution is transferred fewer times between the enclave and Atlas's unprotected environment. However, the encryption process for the enclave's inbound data is responsible for increasing the execution time by an order of magnitude. Also, we notice that increasing the read data buffer beyond 2KB has little to no effect in further increasing the performance. Furthermore, the overall end-to-end execution time of Atlas is increased compared to the vanilla QuickJS engine due to Atlas's required boot time as in this case the system is re-initialized from scratch upon each execution.

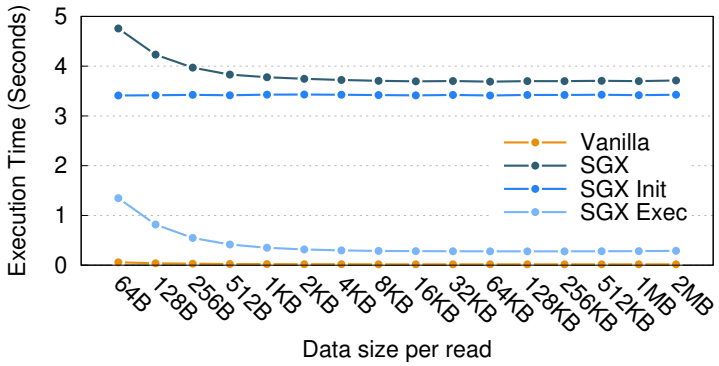


Figure 9.7: Performance comparison between the vanilla QuickJS and Atlas when reading a 20MB file with variable read buffer sizes.

Memory Access Performance

We proceed the evaluation of the enclave's performance properties by measuring the overhead introduced by SGX when randomly accessing memory spaces, performing read and write operations. For this reason, we develop a simple benchmark that performs 1 million random writes to consecutive memory spaces, ranging from 64B to 2MB and afterwards, 1 million random reads to the same locations. We execute this microbenchmark using both the vanilla QuickJS engine and Atlas, where the memory accesses are performed on protected memory. The results of this analysis are presented in Figure 9.8. As we can see, the memory access times achieved by Atlas, both for read and write operations are almost identical to those achieved by the vanilla QuickJS runtime. We also notice the same cache effects for both protected and unprotected environments with a small performance degradation starting to appear at 2MB due to the encrypted memory, also reported by [181].

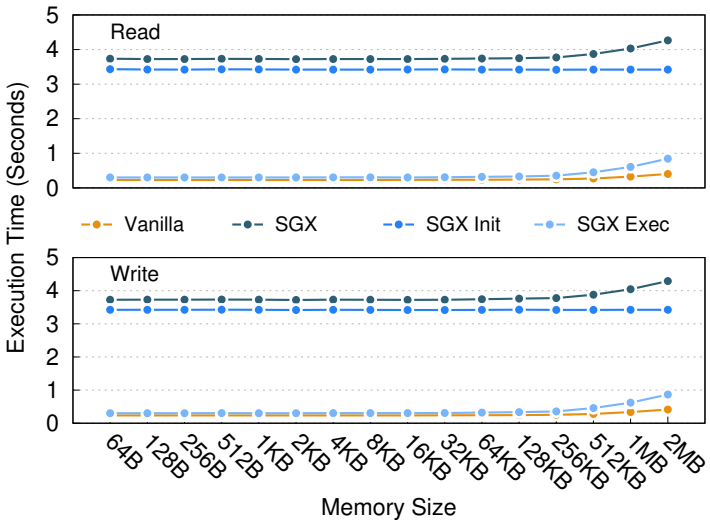


Figure 9.8: Performance comparison between the vanilla QuickJS and our SGX-enabled runtime when performing one million random accesses (read/write) to memory locations ranging between 64B and 2MB.

QuickJS's Default Memory Usage

The final step is to measure the memory footprint of the QuickJS interpreter. To achieve this, we execute a plain JavaScript script and measure the memory usage and object count using the `JS_ComputeMemoryUsage()` and `JS_DumpMemoryUsage()` functions. In this way, we are able to get a view of QuickJS's initialization memory footprint. QuickJS has very minimal memory requirements, allocating only 23KB of memory and 2975 objects, with each

object sized at an average of 8B. Totally, the runtime uses 305KB of memory and 3203 objects. These characteristics render QuickJS an ideal runtime to be encapsulated within SGX enclaves as its minimal memory requirements do not violate the 128MB protected live memory restriction and provides enough headroom for script execution without triggering SGX's expensive page swapping mechanism even on SGXv1 hardware.

9.5 Summary

In this chapter, we presented Atlas, our framework's final component that enables us to automatically scaleout JavaScript components on TEEs. The system uses program transformations to offload function calls of a given application and distribute load among TEE nodes, with respect to the original sequential execution. A modified TEE-embedded language runtime environment, namely QuickJS, and a set of optimizations complete the picture, supporting the secure execution of offloaded code fragments. Atlas's evaluation shows success at TEE-scaleout of legacy applications, not initially developed for TEEs, substantial speedups over naive decomposition, and attractive elasticity characteristics, all achieved with minimal developer effort. This work's contributions are the following:

- We design and implement a secure JavaScript runtime that allows distributed secure execution of new and legacy JavaScript applications with minimal developer effort.
- We present how Atlas successfully offloads secure computations for a wide set of real-world applications and benchmarks using custom and commercial infrastructure.
- We show that user-level enclave-based TEEs can be utilized to perform secure and private distributed computations, providing a significant performance improvement.

9.5.1 Discussion

SGX Enclave Constraints As mentioned in previous chapters, Intel SGX version 1 has a memory limitation of 128MB live encrypted memory. Exceeding this memory cap results in triggering the SGX EPC swap mechanism in Linux. In such cases, Intel's Memory Encryption Engine starts encrypting the trusted pages and storing them to the untrusted DRAM before fetching new ones. Consequently, developers have to efficiently handle data management since memory-bound scripts and functions can still execute on Atlas, but the overhead introduced might be noticeable. In such cases, it is optimal to utilize our system on SGX2 hardware.

Native Module Support As described in Section 9.3.1, native module support is not enabled on Atlas. Dynamically loading and executing shared object libraries require using the `dl*` family of functions. Since these functions need to dynamically allocate and map

the memory as executable, functionality which is not supported on SGX version 1. By offloading such unsupported requests to the untrusted part of the Atlas workers, we can indeed enable and execute shared libraries. However, the mapping process will take place in the untrusted DRAM, resulting in exposing the dynamic library to a potential attacker that could monitor or tamper its execution. Also, systems that enable binary compatibility for SGX enclaves can be possibly utilized to provide secure native module support [173].

Chapter 10
Related Work

In this chapter, we present the research works found in the literature, related to the various modules composing our framework, as well as works that aim to advance the SGX technology or mitigate its limitations.

10.1 Malware Detection

As expected, the research community in the area of security focuses on malware detection to protect users and organizations from the cyber threats that are continually evolving. ClamAV [6] is the most popular open-source anti-malware solution, heavily based on malware detection through pattern matching. CloudAV [137] was one of the first works to put forward the notion of cloud-based malware scanning while [138] extends CloudAV to the mobile environment. Although CloudAV achieves high detection rates, it exposes sensitive information without preserving the user's privacy. SplitScreen implements a distributed anti-malware system to speed up the malware scanning using bloom filters [56]. RScam is another cloud-based anti-malware system which provides an efficient security service and data privacy protection for resource-constrained devices [180]. Still, RScam assumes a trusted server environment. In this work, we propose a cloud-based malware detection module using hardware-assisted enclaves to shield user data and preserve their privacy. CloudNet is a GPU-accelerated anti-malware engine for cloud services [95]. While some works focus on improving the performance of malware detection systems (e.g., [193]), in this work we tackle the urge for strong privacy-preserving guarantees when it comes to computational offloading to untrusted environments, where users have no control over the manipulation of their personal data.

Hand-held devices contain a lot of personal data, as for instance pictures or transcripts, making them a promising target for frauds [158, 175]. Some of the most popular commercial applications on mobile malware detection are the AVG Antivirus [4] and Avira mobile security [5] solution. Google provides its own service, namely Google Play Protect, as a built-in application that automatically scans and verifies installed Android ap-

plications [8]. Unlike our proposed system, this solution is destined only for Android applications retrieved from Google's Play Store. Similar research works, based on signature scanning, proposed systems that aim for low power consumption [107, 126]. Other techniques for malware detection are code and API call analysis, used in ScanMe Mobile [217] and [30] respectively. In Paranoid Android [153], security checks are applied on remote security servers that host exact virtual replicas of devices, applying multiple detection techniques simultaneously. Other works, utilize machine learning techniques to apply malware detection on mobile devices [31, 210]. In general, machine learning techniques for malware detection have recently gained increased popularity, with multiple works utilizing various models and behavioural analysis to identify malicious code samples [33, 42, 70, 98, 177]. However, these works do not focus on real-time protection. Finally, we have previously proposed a GPU-assisted antivirus solution on Android devices through edge offloading [72]. While our work is based on the same grounds (i.e., the malware detection with support for mobile devices), we advance the state-of-the-art by proposing a practical, yet secure, malware detection engine that strongly focuses on preserving user privacy, offloading the processing of personal data inside a trusted execution environment.

10.2 Kernel Integrity Monitoring

Kernel integrity monitors can be divided into two main categories, (i) software based and (ii) hardware based, with the former often relying on a hypervisor. Azab et al. [44] proposed a system, similar to our design, that leverages ARM TrustZone [39] to perform kernel monitoring. The benefit of this approach is that the mappings and the virtual address translations of the kernel are transferred directly inside the trusted application, rendering the normal world unable to view or modify them. However, this model is only applicable to TrustZone-capable devices such as IoT and mobile phones. HyperCheck [201] is a hardware-assisted tampering detection framework that leverages the CPU System Management Mode (SMM) and aims to protect the integrity of VMMs and the host's OS against certain classes of attacks. In contrast to our system, where the analysis is performed in the secure enclave executed on the target system, HyperCheck transmits the entire system state to an external server. Feng et al. introduce BehaviorKI [82], a behavior-triggered integrity checking system that extracts a set of behavior patterns by analyzing attacking processes and triggers checking with kernel invariants. OCsk [97] is a virtual machine based approach able to detect rootkits by determining violations to operating system invariants. Also, SecVisor [167] is a hypervisor based approach that protects the kernel's integrity by verifying that only user-approved code can execute in kernel mode, thus protecting the OS against code injection attacks and rootkits.

Copilot [148] is a snapshot based kernel integrity monitor that, similar to our proposal,

requires no modifications to the monitored system but is implemented on a custom FPGA. Moon et al. proposed Vigilare [130], a hardware- and snooping-based integrity monitor that scans the bus for memory accesses that can possibly affect the kernel's security. Also, KI-Mon [121] extends Vigilare by offering event-triggered protection for mutable objects. Hypernel [119] is a security framework that combines a software and a hardware component, enabling a word-granularity monitoring capability on the kernel memory. Finally, Koromilas et al. proposed GRIM [112], an external kernel integrity monitor that performs snapshot-based scanning by utilizing an external GPU. Our proposed module is able to successfully monitor the underlying OS kernel without the need for external hardware or hypervisor, residing in the user space in a protected enclave, while it can be used in combination with the rest of our stack's components.

Similar to our approach, EPA-RIMM [71] periodically verifies the kernel's integrity by checking the SHA-256 hash values of specific memory regions, control registers, and model-specific registers but does so by leveraging the System Management Mode (SMM). Win et al. [207] suggested the utilization of only 8 bytes by hashing from the initial starting offset of the 9th byte, aiming to reduce performance overheads. Also, in CloudMon [205], the authors utilize system call addresses and system call hash values to detect kernel-level rootkits in cloud environments. Finally, SBCFI [149] is a system using state-based control flow integrity that also utilizes a hash function to validate the kernel text, including static control flow transfers.

10.3 Trusted Execution for Android

In the area of mobile devices, ARM TrustZone [39] is the most popular TEE which enables the development of two separate environments, the *trusted* and the *untrusted* world. This split enables the execution of the rich OS (that runs in the untrusted world) and the system software that controls basic operations that must be protected and runs in the trusted world. Santos et al. [161] use TrustZone for securing mobile applications by establishing and isolating trusted components. However, their approach requires a trusted language runtime in the TCB, due to the fact that there is only a single trusted world. DroidVault [123] presents a security solution for storing and manipulating sensitive data. The data are stored in an encrypted format on the file system and are only processed (decrypted) in TrustZone. TZ-RKP implements a low-TCB system-level safe security monitor on top of the TrustZone architecture [44] that provides a real-time OS kernel protection. The monitor routes privileged system functions through secure world for examination. Samsung KNOX [160] is a secure container framework, leveraging ARM TrustZone, that offers protection from both the software and the hardware. However, KNOX is primarily a closed-source system available only to flagship Samsung devices and its internals and APIs are not documented in the open literature. A major limitation of the proposed TrustZone-

based systems, which our solution is able to address, is their inability to provide secure containers on-demand to support the development of new applications. Also, these approaches fail to protect against attackers with physical DRAM access. Moreover, TrustZone is not best suited to be securely shared by multiple applications, as there is only one shared TEE provided by the hardware, offering limited or no isolation granularity between applications compared to SGX. This prevents it from being leveraged simultaneously by multiple applications, either in user space (e.g., banking, medical applications, etc.) or kernel-space (e.g., security monitors, device keystore, etc.).

In the context of safeguarding common services offered by operating systems, Kumar et al. [116] proposed SecureFS, a secure file system leveraging Intel SGX enclaves. Also, Peters et al. introduced BASTION-SGX [147], a system that enables trusted I/O at the architectural level for the Bluetooth stack. SGXIO [203] is another work that enables support for trusted paths to generic I/O devices by combining SGX's features and a programming model with traditional hypervisor-based trusted path architectures.

Haven [49] aims to execute unmodified legacy Windows applications inside SGX enclaves by porting a Windows library OS into SGX. Graphene-SGX [188] 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. VC3 [164] uses SGX to achieve confidentiality and integrity for the Map Reduce framework. Also, SCONE [41] is a shielded execution framework that enables developers to compile their C applications into Docker containers.

In contrast to these works, we propose the first approach, to the best of our knowledge, that enables native SGX enclaves for the Android OS. Moreover, there are recently proposed approaches that implement user-level enclaves, similar to SGX, either independent of the underlying CPU [83] or on top of ARM TrustZone [55]. Recently, Zhao et al. proposed vSGX [219], a system that aims to virtualize SGX execution in SEV-enabled AMD processors, enabling the execution of SGX enclaves through instruction emulation and integration with SEV-based memory protection. Our proposed Android services are not fundamentally tight to Intel SGX and, as such, could be implemented on top of such approaches instead, provided that they can be ported transparently across multiple Android-based devices.

10.4 Secure and Attested End-to-End Communication

Many applications and recent research utilize Intel SGX, a TEE that has been integrated in many large-scale projects that have needs for increased security. SGX has been used to enhance the Snort IDS [117] and the TOR network [109], while efforts have also been made to move TLS endpoints inside SGX enclaves [110] and provide SGX-enabled VPN services [145]. More related to our work, Balfe et al. [46] have shown how TPMs can be used

in peer-to-peer networks to provide security. In their work, they use the Trusted Computing technology to establish pseudonymous identifiers and build secure channels between peers. Additionally, mutual attestation with TPMs has also been used in creating protocols for trusted RFID systems [132]. Shepherd et al. [168] raise the challenge of secure TEE-to-TEE communication between remote sensing devices. Most recently, Ghaffar et al. [87] proposed an improved model to achieve data privacy when handling in cloud environments using proxy re-encryption. Also, [73] utilizes remote attestation in the proximity verification process to enable secure connections between enclaves and trusted embedded devices. In [43], the authors proposed TaLoS, a library that aims to replace standard TLS libraries by providing connection endpoints inside SGX enclaves. Similar approaches are also followed by other works [11, 14, 28], aiming to integrate SGX in SSL.

Furthermore, Intel SGX technology has been used for secure many-party applications [114] and secure multiparty computations [45], where the enclaves act as middleboxes. For example, ShieldBox [186] utilizes Intel SGX to provide secure middleboxes for high performance network functions that can be deployed on untrusted servers. Our work can be applied on middleboxes to provide communication using mutually trusted channels.

Based on TrustZone, SANCTUARY [55] is a system that enables the execution of security-sensitive applications within strongly isolated compartments. These compartments are mutually distrusted and reside within ARM's TrustZone world, thus being comparable to SGX's enclaves. Their framework offers attestation service to a third party using a *Proxy Sanctuary Application* which is comparable to Intel's Quoting Enclave. Our work could also be applied to the model offered by SANCTUARY.

10.5 Trusted Dynamic Code Execution

Since SGX's initial release, many works aim to provide dynamic trusted code execution using enclaves as well as execution of legacy applications. In this area, VC3 [164] uses SGX to achieve confidentiality and integrity only for the Map Reduce framework. Haven [49] aims to execute unmodified legacy Windows applications inside SGX enclaves by porting a Windows libOS into SGX. Similarly, Graphene-SGX [188], later evolved to Gramine [9], encapsulates an 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 [41] is a shielded execution framework that enables developers to compile their C applications into Docker containers. These works are either domain-specific, provide a container-based, or a libOS-based approach in which users can run general-purpose applications. Our work provides a balance between these approaches by providing general purpose program execution through high-level programming languages, such as Lua and JavaScript that also keeps the TCB size minimal and highly optimized, contrary to Docker- or libOS-based approaches.

Ryoan [100] provides a distributed framework that utilizes SGX enclaves to protect sandbox instances, written in C, from potentially malicious computing platforms. It is designed so that confined modules operate on the given input once and do not hold their state (similar to our work) preventing potential data and state leakage. Glamdring [125] is a source-level partitioning framework that secures applications written in plain C. The developer has to explicitly 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 our proposed secure execution modules, the developer has to recompile the code with the newly added annotations and then perform the analysis to partition the code. On our security stack, additional annotations are optional and used to achieve isolation granularity when explicitly needed.

Similar to our work, Civet [189] is a framework developed concurrently with our system and partitions Java applications into SGX enclaves. The framework also provides garbage collection but introduces significantly greater overhead compared to our proposal. TrustJS [90] also explores the possibility of bridging JavaScript with SGX enclaves to protect security-sensitive JavaScript components inside browser applications. However, TrustJS requires script partitioning for the trusted code segments, data, and meta-information. Additionally, it does not attempt to provide a general-purpose execution sandbox for legacy JavaScript applications. ScriptShield [199] enables development in SGX enclaves using high-level scripting languages. In contrast to our work though, it fails to demonstrate its practicality on real-life applications in terms of performance. Also, none of the discussed systems enables secure distributed execution, utilizing multiple trusted computation nodes. [96] introduces a reactive middleware framework approach for data stream processing on the cloud based on Intel SGX. Overall, our solution builds on the same objectives and also provides many optimizations that are necessary to make it practical in terms of performance and multiple concurrent instance execution.

Aiming to protect high-level code execution, Ghosn et al. introduced GOTEE [88], a backward-compatible fork of the Go compiler. The system, tries to extend Golang functionality by enabling trusted execution of `goroutines` inside an SGX enclave, relying solely on the compiler to extract the source code and the data. RUST-SGX [200] provides an SGX SDK extension to enable enclave utilization from applications developed in Rust while [75] enables secure compartmentalization of Rust-based applications. Focusing on JIT compilers, JITGuard [84] leverages enclaves to provide a trusted environment for emitting dynamic code, thus rendering it tamper-proof. Finally, SGXPy [215] introduces an integrity preserving tool for Python applications utilising SGX and a library OS.

The research community has also proposed various systems to protect Java-based applications using SGX. Montsalvat [213] utilizes enclaves along with the GraalVM and code annotations to protect sensitive code segments. Also, many other works leverage annotations to generate smaller TCBs and protect critical code and data [81, 135, 136].

10.6 Remote Confidential Computations

As the need for lower costs, higher performance and scalability rises, outsourcing network processing applications to the cloud has become tempting. APLOMB is a service for outsourcing enterprise middlebox processing to the cloud [169]. To provide confidentiality, BlindBox proposes the processing of encrypted traffic [170], something that leads to unpractical computational requirements. Likewise, Embark, aiming to offer security and confidentiality, enables a cloud provider to support middlebox functionality by processing encrypted traffic [120]. A common service that cloud providers offer is storage, yet, there is no transparency over the manipulation of user data [133, 191, 198, 209]. CloudFence provides transparent data tracking capabilities to both service providers and users [144].

As discussed, TEEs, such as Intel SGX, can guarantee data and code protection. Thus, many works focus on the exploitation of this technology for applications outsourced in the cloud. For instance, VC3 [164], Opaque [220] and [58] offer privacy-preserving data analytics in the cloud using Intel SGX. In addition, EndBox [89], ShieldBox [186] and SafeBricks [152] focus on securing middlebox functionality using Intel SGX.

The data protection mechanisms provided by trusted execution environments, such as Intel SGX, have been widely utilized in the context of securing databases. EnclaveDB [154] is a database engine that can guarantee confidentiality, integrity, and freshness for data and queries. StealthDB [196] also proposed an SGX-based database system with a minimal TCB. A similar approach has also been followed by VeriDB [221] that further leverages enclaves to provide additional data verification. Also, Azure utilizes enclaves to protect sensitive data stored in the Azure SQL Database, Azure SQL Managed Instance, and SQL Server database [37] while ProDB [94] proposes and architecture that utilizes both SGX and oblivious RAM to provide database security.

While there are works that enable the execution of unmodified applications in enclaves (e.g., [41, 49, 174, 185, 188]), we choose not to follow such approach (i.e., execute our modules on top of SGX using one of the aforementioned tools) since these systems result to an increased trusted computing base with many dependencies, which widens the attack surface [52, 162] and significantly decreases the end-to-end performance.

SGX is also heavily utilized in the context of blockchain applications. One of the earliest work in this field, named Town Crier [216] introduces SGX's utilization in bridging smart contracts and known websites, handling authentication and user credentials. Also, a plethora of other works focus on handling private contracts [61, 80, 122, 124, 202, 212], leveraging SGX's capabilities. Furthermore, SGX has been exploited by many works in the field of digital rights management, data streaming and licensing [47, 60, 63, 85, 115, 165]. Finally, Madsen et al. [127] transform general omission resilient algorithms into Byzantine fault-tolerant algorithms using SGX.

10.7 Improving Intel SGX

Several improvements for SGX have been recently developed to protect it against memory bugs [166] or controlled-channel attacks [171]. SGXBOUNDS [118] enables bounds checking with low memory overheads to fit within the limited EPC size. SGX-Shield [166] implements Address Space Layout Randomization (ASLR) in enclaves, with a scheme to maximize the entropy, and the ability to hide and enforce ASLR decisions. Eleos [142] proposed to reduce the number of enclave exits by asynchronously servicing system calls outside of the enclaves and enabling user space memory paging. Also, SGX-Elide [48] aims to protect the secrecy of SGX code itself by enabling dynamic updates of the enclave code. T-SGX [171] is an approach that combines SGX with Transactional Synchronization Extensions to mitigate controlled-channel attacks. All these works are orthogonal to our security stack and can be integrated to our proposed framework.

Furthermore, SGXPecial [129] introduces an interface specialization tool, extending the `Edger8r` SGX SDK tool, that performs API specialization at build time thus restricting the valid control flows between the host and the enclave. Also, [131] proposed a technique to attest the enclave runtime and enables a remote verifier to assess its integrity. Shimizu et al. [172] aim to reduce the overheads introduced by SGX's cryptographic operations with a cache replacement system, called Ea-plru, that favors an enclave cache line over a non-enclave cache line. Our framework also utilizes caching systems to improve the enclave's performance in multiple cases (e.g., DFA-state caching, RA response caching, etc.).

Aiming to improve SGX's performance, Weisse et al. [204] introduce a system that leverages shared memory and enclave thread bound to an untrusted thread using spinlocks to synchronize the communication between the two contexts, while VAULT [183] extends this approach enabling multiple enclave function execution. Also, focusing on reducing the overhead introduced by frequent `OCALL`/`ECALL` context switches, [108, 184] and [214] propose techniques for optimizing switchless calls.

Chapter 11
Conclusion

In conclusion, this work has demonstrated the potential of user-level enclave-based Trusted Execution Environments (TEEs) in developing a modular security framework for desktop and mobile systems. The framework consists of interoperable components that safeguard various software layers and prioritize user privacy, operating seamlessly with off-the-shelf software and hardware infrastructure.

Aiming to address multiple security concerns, the framework is divided into four layers: (i) host protection, (i) operating system service protection, (iii) secure communications, and (iv) secure dynamic code execution. The framework encompasses six vital components: (i) a malware detection engine, (ii) a kernel integrity monitor, (iii) a secure and attested end-to-end communication service, (iv) a TEE-enhanced Android OS, (v) two protected high-level language runtimes for secure dynamic code execution, and (vi) a scalable secure execution system utilizing multiple TEE-enabled worker nodes.

Key outcomes of this research include a signature-based malware detection solution, capable of local or remote deployment without compromising user privacy. The system performs rule-based malware detection using our lightweight pattern detection engine, providing a minimal TCB, and a set of caching systems that enable secure pattern matching even with memory-restricted TEEs. Additionally, a lightweight integrity monitor that leverages user-level enclaves to remain secure and undetected in user space, scanning memory regions that are expected to be immutable.

Also, the security stack extends to the Android platform through our custom port of the Intel SGX SDK, PSW, driver, and required services. This results in the development of the first SGX-enabled Android OS, providing TEE-protected services, simple APIs for enclave utilization and a custom cross-compiler toolchain for seamless deployment on a diverse set of mobile devices without intrusive device modifications. Moreover, our SGX-enabled handshake protocol and RA response caching system allows for enclave-to-enclave secure and attested network communications.

Finally, the framework aims to tackle common limitations of existing TEE technologies that hinder their widespread adoption in modern software ecosystems built with high-

level, type-safe languages. By treating code as data and leveraging two custom ported language runtimes, namely LuaVM and QuickJS, the framework offers two modules that provide dynamic software execution within enclaves.

In summary, this research presents a complete security framework that leverages user-level enclave-based TEEs to provide robust protection, privacy, secure and attested communications, as well as dynamic execution capabilities across desktop and mobile systems. The advancements made in malware detection, kernel integrity monitoring, secure communication, protected language runtimes, and enclave utilization lay a solid foundation for enhancing system security and promote the wider adoption of TEE technologies in modern software ecosystems. A list of publications produced so far by the activities related to this dissertation is presented in Appendix A.

11.1 Synopsis of Contributions

The main contributions of this work cover the areas of (i) *host protection*, (ii) *operating system service protection*, (iii) (iv) *secure communications*, and (v) *secure dynamic code execution*, and are summarized as follows:

- We design and implement a practical malware detection system that provides users with strong privacy-preserving guarantees regarding the processing of their personal and sensitive data remotely, utilizing hardware assisted enclaves.
- We propose techniques to mitigate the memory constraints introduced by modern TEEs that affect signature-based analysis solutions, such as malware or intrusion detection systems.
- We implement an integrity monitor that leverages user-level enclaves to protect its code and data from identification and modification, residing in the monitored host.
- We demonstrate the effectiveness of the proposed system in identifying transient kernel-side attacks and explore the appropriate monitoring intervals that guarantee that transient rootkits are always detectable and ineffective.
- We describe a systematic methodology for porting the SGX framework to the Android OS and the development of the required cross-compiler toolchain. This methodology can be used as a baseline for porting SGX to other officially unsupported platforms.
- We develop the first, to our knowledge, SGX-enabled Android OS, providing multiple SGX-enhanced Android services and APIs that enhance the system's security and enable externally paired devices to utilize TEE technologies.

- We design and develop a system that provides seamless establishment of enclave-to-enclave (CPU-to-CPU) attested and encrypted network communication between two or more parties.

- We extend our secure communication protocol with a caching system for RA responses that renders it comparable to a standard TLS handshake.

- We implement two lightweight secure execution runtimes that enable confidential computing using high-level type-safe languages, namely Lua and JavaScript, eliminating the need to port code to device-specific TEEs.

- We enhance the secure JavaScript runtime with a scheduling middleware and a tag interpreter that enable the secure distributed execution of selected functions across multiple server nodes hosting instances of TEE-protected runtimes.

- We provide thorough evaluation that proves that our protected runtimes can be used to securely execute and distribute a diverse set of new and legacy applications with little-to-zero code modifications.

11.2 Directions for Future Work

In previous chapters, we have discussed the limitations of each component as well as possible solutions or partial directions for future work. In this section we summarize the steps we can follow to improve our framework, extend its capabilities, or repurpose the functionality of some of our framework's components.

The first and most obvious direction for our future work is to utilize SGX2 hardware and exploit its capabilities. This will allow us, with specific modifications applied on some components, to lift existing encrypted memory constraints and further improve the performance of most modules. For example, we will be able to process more malware patterns simultaneously, provide our custom attestation infrastructure or execute memory-bound high-level applications with lower overheads. Also, the multi-threading capabilities and dynamic memory management provided by SGX2 are expected to further improve our framework's performance and allow us to internally parallelize most modules' execution.

An other direction we wish to explore is utilizing the secure execution environments, namely LuaGuardia and Atlas, to develop a wide set of security tools and general purpose libraries and applications. These environments can be used to develop traffic analysis toolkits, complex firewall and VPN solutions, PKI and authentication services, as well as cryptographic libraries, that could directly be used by new and existing software developed in Lua or JavaScript. Also, our secure runtimes can be extended to build a set of protected distributed black-box systems that serve as building blocks for secure and private

remote execution, providing general purpose services, such as analytics, NLP and sentiment analysis, etc. Moreover, our ecosystem could provide secure drop-in replacements for popular modules that could benefit from private and secure execution, especially in remote execution environments.

Finally, we wish to leverage our extended knowledge on porting complex systems in user-level enclaves to design, from scratch, a secure JavaScript runtime specifically optimized for execution within TEEs. We wish to further extend such a secure runtime with techniques to automatically identify which components need to be securely offloaded and aim towards enabling transparent protected scaleout of complex existing platforms.

Bibliography

[1] A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[2] Android Services. https://developer.android.com/guide/components/services.html.

[3] Android trusty tee. https://source.android.com/security/trusty.

[4] AVG AntiVirus for Android. https://www.avg.com/en-eu/antivirus-for-android.

[5] Avira: Download security, privacy, and speed-enhancing apps for Android and iOS. https://www.avira.com/en/mobile-security.

[6] ClamAV — Cisco Talos Intelligence Group. https://www.talosintelligence.com/clamav.

[7] Crystax NDK. https://www.crystax.net/android/ndk/.

[8] Google Play Protect. https://www.android.com/play-protect/.

[9] Gramine - a Library OS for Unmodified Applications. https://gramineproject.io/.

[10] Intel SGX2 Support Coming With Linux 6.0. https://www.phoronix.com/news/Intel-SGX2-Linux-6.0.

[11] Intel Software Guard Extensions SSL. https://github.com/intel/intel-sgx-ssl.

[12] iperf, the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/.

[13] Lua 5.3 Reference Manual. https://www.lua.org/manual/5.3/.

[14] mbedtls-SGX: a TLS stack in SGX. https://github.com/bl4ck5un/mbedtls-SGX.

[15] Mobile Operating System Market Share Worldwide, May 2022 - May 2023. https://gs.statcounter.com/os-market-share/mobile/worldwide.

[16] Modest Natural Language Processing. https://github.com/spencermountain/compromise.

[17] Open enclave sdk. https://openenclave.io/sdk/.

[18] Open portable trusted execution environment. https://www.op-tee.org/.

[19] Packet filtering in Lua. https://github.com/Igalia/pflua.

[20] PFlang. https://github.com/Igalia/pflua/blob/master/doc/pflang.md.

[21] Quickjs javascript engine. https://bellard.org/quickjs/.

[22] RFC 6238, TOTP: Time-Based One-Time Password Algorithm. https://www.ietf.org/rfc/rfc6238.txt.

[23] Signal, Speak Freely. https://signal.org/en/.

[24] Snabb: Simple and fast packet networking. https://github.com/snabbco/snabb.

[25] Snort - network intrusion detection & prevention system. https://www.snort.org/.

[26] sysvipc(7), Linux Manual Page. https://man7.org/linux/man-pages/man7/sysvipc.7.html.

[27] tcpdump. https://www.tcpdump.org/manpages/tcpdump.1.html.

[28] wolfSSL with Intel SGX. https://www.wolfssl.com/wolfssl-with-intel-sgx/.

[29] Trusted little kernel (tlk) for tegra: Foss edition. http://nv-tegra.nvidia.com/gitweb/?p=3rdparty/ote_partner/tlk.git;a=blob_plain;f=documentation/Tegra_BSP_for_Android_TLK_FOSS_Reference.pdf;hb=HEAD, 2015.

[30] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[31] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glauco Barroso Junquera, and Paulo Lício de Geus. Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17, 2015.

[32] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[33] Muhammad Ali, Stavros Shiaeles, Gueltoum Bendiab, and Bogdan Ghita. Malgra: Machine learning and n-gram malware feature extraction and detection system. *Electronics*, 9(11):1777, 2020.

[34] Tiago Alves. Trustzone: Integrated hardware and software security. *White paper*, 2004.

[35] AMD. SEV Secure Nested Paging Firmware ABI Specification. `https://www.crystax.net/android/ndk/`.

[36] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[37] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. Azure sql database always encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1511–1525, 2020.

[38] Apple. Apple Secure Enclave. `https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web`.

[39] ARM LIMITED. ARM Security Technology - Building a Secure System using TrustZone Technology.

[40] Md Armanuzzaman and Ziming Zhao. Byotee: Towards building your own trusted execution environments using fpga. *arXiv preprint arXiv:2203.04214*, 2022.

[41] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[42] Muhammad Asam, Shaik Javeed Hussain, Mohammed Mohatram, Saddam Hussain Khan, Tauseef Jamal, Amad Zafar, Asifullah Khan, Muhammad Umair Ali, and Umme Zahoora. Detection of exceptional malware variants using deep boosted feature spaces and machine learning. *Applied Sciences*, 11(21):10464, 2021.

[43] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. Talos: Secure and transparent tls termination inside sgx enclaves. 2017.

[44] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102, 2014.

[45] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from sgx. In *International Conference on Financial Cryptography and Data Security*, 2017.

[46] Shane Balfe, Amit D Lakhani, and Kenneth G Paterson. Trusted computing: Providing security for peer-to-peer networks. In *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, 2005.

[47] Erick Bauman and Zhiqiang Lin. A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, pages 1–6, 2016.

[48] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. Sgxelide: enabling enclave code secrecy via self-modification. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 75–86, 2018.

[49] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[50] Fabrice Bellard. QuickJS Benchmark. `https://bellard.org/quickjs/bench.html`.

[51] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. Tweetnacl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*, pages 64–83. Springer, 2014.

[52] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel {SGX}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1213–1227, 2018.

[53] Rick Boivie and Peter Williams. Secureblue++: Cpu support for secure executables. *Yorktown Heights, NY, USA: IBM, Tech. Rep. RC25287*, 2013.

[54] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[55] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[56] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G Andersen. Splitscreen: Enabling efficient, distributed malware detection. In *NSDI*, pages 377–390, 2010.

[57] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[58] Swarup Chandra, Vishal Karande, Zhiqiang Lin, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. Securing data analytics on sgx with randomization. In *European Symposium on Research in Computer Security*. Springer, 2017.

[59] Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, and Farinaz Koushanfar. Deepattest: An end-to-end attestation framework for deep neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 487–498. IEEE, 2019.

[60] Jian Chen, Bo Dai, Yanbo Wang, Yiyang Yao, and Bo Li. Sectube: Sgx-based trusted transmission system. In *Smart Computing and Communication: Second International Conference, SmartCom 2017, Shenzhen, China, December 10-12, 2017, Proceedings 2*, pages 231–238. Springer, 2018.

[61] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

[62] U Cisco. Cisco annual internet report (2018–2023) white paper. *Cisco: San Jose, CA, USA*, 10(1):1–35, 2020.

[63] Tobias Cloosters, Sebastian Surminski, Gerrit Sangel, and Lucas Davi. Salsa: Sgx attestation for live streaming applications. In *2022 IEEE Secure Development Conference (SecDev)*, pages 45–51. IEEE, 2022.

[64] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal De Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–189, 2015.

[65] Debra L Cook, Ricardo Baratto, and Angelos D Keromytis. Remotely keyed cryptographics secure remote display access using (mostly) untrusted hardware. In *International Conference on Information and Communications Security*, pages 363–375. Springer, 2005.

[66] Intel Corporporation. Intel 64 and ia-32 architectures software developer manuals, 2018.

[67] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[68] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 857–874, 2016.

[69] Aimee Coughlin, Greg Cusack, Jack Wampler, Eric Keller, and Eric Wustrow. Breaking the trust dependence on third party processes for reconfigurable secure hardware. In *Proceedings of the 2019 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 282–291, 2019.

[70] SL Shiva Darshan and CD Jaidhar. An empirical study to estimate the stability of random forest classifier on the hybrid features recommended by filter based feature selection technique. *International Journal of Machine Learning and Cybernetics*, 11:339–358, 2020.

[71] Brian Delgado and Karen L Karavanic. Epa-rimm: A framework for dynamic smm-based runtime integrity measurement. *arXiv preprint arXiv:1805.03755*, 2018.

[72] Dimitris Deyannis, Rafail Tsirbas, Giorgos Vasiliadis, Raffaele Montella, Sokol Kosta, and Sotiris Ioannidis. Enabling gpu-assisted antivirus protection on android devices through edge offloading. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*, pages 13–18. ACM, 2018.

[73] Aritra Dhar, Ivan Puddu, Kari Kostiainen, and Srdjan Capkun. Proximitee: Hardened sgx attestation by proximity verification. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pages 5–16, 2020.

[74] Android Documentation. The android keystore system. `https://developer.android.com/training/articles/keystore`.

[75] Felix Dreissig, Jonas Röckl, and Tilo Müller. Compiler-aided development of trusted enclaves with rust. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–10, 2022.

[76] Brendan Eich. ECMAScript 6. `http://es6-features.org/`.

[77] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *Ndss*, volume 12, pages 1–15, 2012.

[78] Evervault. What is a trusted execution environment (tee)? `https://evervault.com/blog/what-is-a-trusted-execution-environment-tee`.

[79] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 190–202. IEEE, 2014.

[80] Min Fang, Zhao Zhang, Cheqing Jin, and Aoying Zhou. An sgx-based execution framework for smart contracts upon permissioned blockchain. *Distributed and Parallel Databases*, pages 1–36, 2022.

[81] Anter Faree and Yongzhi Wang. Protecting security-sensitive data using program transformation and intel sgx. In *2019 International Conference on Networking and Network Applications (NaNA)*, pages 421–428. IEEE, 2019.

[82] Xinyue Feng, Qiusong Yang, Lin Shi, and Qing Wang. Behaviorki: Behavior pattern based runtime integrity checking for operating system kernel. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018.

[83] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305, 2017.

[84] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Jitguard: hardening just-in-time compilers with sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2405–2419, 2017.

[85] Benny Fuhry, Lina Hirschoff, Samuel Koesnadi, and Florian Kerschbaum. Segshare: Secure group file sharing in the cloud using enclaves. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 476–488. IEEE, 2020.

[86] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206, 2003.

[87] Zahid Ghaffar, Shafiq Ahmed, Khalid Mahmood, Sk Hafizul Islam, Mohammad Mehedi Hassan, and Giancarlo Fortino. An improved authentication scheme for remote data access and sharing over cloud storage in cyber-physical-social-systems. *IEEE Access*, 8:47144–47160, 2020.

[88] Adrien Ghosn, James R Larus, and Edouard Bugnion. Secured routines: Language-based construction of trusted execution environments. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 571–586, 2019.

[89] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, et al. Endbox: Scalable middlebox functions using client-side trusted execution. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 386–397. IEEE, 2018.

[90] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. Trustjs: Trusted client-side execution of javascript. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[91] Johannes Götzfried, Tilo Müller, Ruan De Clercq, Pieter Maene, Felix Freiling, and Ingrid Verbauwhede. Soteria: Offline software protection within low-cost embedded devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 241–250, 2015.

[92] James Greene. Intel trusted execution technology, white paper. *Online: http://www. intel. com/txt*, 2012.

[93] Trusted Computing Group et al. Tpm main part 1 design principles. *TCG White Paper*, 2011.

[94] Ziyang Han and Haibo Hu. Prodb: A memory-secure database using hardware enclave and practical oblivious ram. *Information Systems*, 96:101681, 2021.

[95] George Hatzivasilis, Konstantinos Fysarakis, Ioannis Askoxylakis, and Alexander Bilanakos. Cloudnet anti-malware engine: Gpu-accelerated network monitoring for cloud services. In *International Workshop on Information and Operational Technology Security Systems*, pages 122–133. Springer, 2018.

[96] Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio 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*, pages 124–133, 2017.

[97] Owen S Hofmann, Alan M Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with osck. *ACM SIGARCH Computer Architecture News*, 39(1):279–290, 2011.

[98] Xiang Huang, Li Ma, Wenyin Yang, and Yong Zhong. A method for windows malware detection based on deep learning. *Journal of Signal Processing Systems*, 93:265–273, 2021.

[99] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud gpus. In *NSDI*, pages 817–833, 2020.

[100] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–32, 2018.

[101] Dongil Hwang, Sanzhar Yeleuov, Jiwon Seo, Minu Chung, Hyungon Moon, and Yunheung Paek. Ambassy: A runtime framework to delegate trusted applications in an arm/fpga hybrid system. *IEEE Transactions on Mobile Computing*, 2021.

[102] R Ierusalimschy, LH Figueiredo, and W Celes. Lua: an extensible embedded language. *Dr. Dobb's Journal*, 21(12):26œ33, 1996.

[103] IDC International Data Corporation. Smartphone market share. `https://www.idc.com/promo/smartphone-market-share`, 2023.

[104] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: Epid provisioning and attestation services. *White Paper*, 1(1-10):119, 2016.

[105] David Kaplan. {AMD} x86 memory encryption technologies. 2016.

[106] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[107] Hahnsang Kim, Joshua Smith, and Kang G Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2008.

[108] Seongmin Kim. An optimization methodology for adapting legacy sgx applications to use switchless calls. *Applied Sciences*, 11(18):8379, 2021.

[109] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. Sgx-tor: A secure and practical tor anonymity network with sgx enclaves. *IEEE/ACM Transactions on Networking*, 26(5), 2018.

[110] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863*, 2018.

[111] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[112] Lazaros Koromilas, Giorgos Vasiliadis, Elias Athanasopoulos, and Sotiris Ioannidis. Grim: leveraging gpus for kernel integrity monitoring. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 3–23. Springer, 2016.

[113] Hugo Krawczyk. Sigma: The 'sign-and-mac' approach to authenticated diffie-hellman and its use in the ike protocols. In *CRYPTO*, 2003.

[114] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N Asokan, Andrew Simpson, and Robin Ankele. Exploring the use of intel sgx for secure many-party applications. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, 2016.

[115] Sandeep Kumar, Abhisek Panda, and Smruti R Sarangi. Securelease: Maintaining execution control in the wild using intel sgx. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, pages 29–42, 2022.

[116] Sandeep Kumar and Smruti R Sarangi. Securefs: a secure file system for intel sgx. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 91–102, 2021.

[117] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort intrusion detection system with intel software guard extension (intel sgx). *arXiv preprint arXiv:1802.00508*, 2018.

[118] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221, 2017.

[119] D. Kwon, K. Oh, J. Park, S. Yang, Y. Cho, B. B. Kang, and Y. Paek. Hypernel: A hardware-assisted framework for kernel protection without nested paging. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2018.

[120] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: securely outsourcing middleboxes to the cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[121] Hojoon Lee, Hyungon Moon, Daehee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 511–526, 2013.

[122] Rujia Li, Qin Wang, Qi Wang, David Galindo, and Mark Ryan. Sok: Tee-assisted confidential smart contract. *arXiv preprint arXiv:2203.08548*, 2022.

[123] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Prateek Saxena. Droidvault: A trusted data vault for android devices. In *2014 19th International Conference on Engineering of Complex Computer Systems*, pages 29–38. IEEE, 2014.

[124] Xueping Liang, Sachin Shetty, Peter Foytik, and Deepak Tosh. Enforcing security and privacy in distributed ledgers by intel sgx. In *2020 Spring Simulation Conference (SpringSim)*, pages 1–12. IEEE, 2020.

[125] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 285–298, 2017.

[126] Lei Liu, Guanhua Yan, Xinwen Zhang, and Songqing Chen. Virusmeter: Preventing your cellphone from spies. In *International Workshop on Recent Advances in Intrusion Detection*, pages 244–264. Springer, 2009.

[127] Mads Frederik Madsen, Mikkel Gaub, Malthe Ettrup Kirkbro, and Søren Debois. Transforming byzantine faults using a trusted execution environment. In *2019 15th European Dependable Computing Conference (EDCC)*, pages 63–70. IEEE, 2019.

[128] Brian McGillion, Tanel Dettenborn, Thomas Nyman, and N Asokan. Open-tee–an open virtual trusted execution environment. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 400–407. IEEE, 2015.

[129] Shachee Mishra and Michalis Polychronakis. Sgxpecial: Specializing sgx interfaces against code reuse attacks. In *Proceedings of the 14th European Workshop on Systems Security*, pages 48–54, 2021.

[130] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 28–37, 2012.

[131] Mathias Morbitzer. Scanclave: verifying application runtime integrity in untrusted environments. In *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 198–203. IEEE, 2019.

[132] Mohd Faizal Mubarak, Saadiah Yahya, et al. Mutual attestation using tpm for trusted rfid protocol. In *2010 Second International Conference on Network Applications, Protocols and Services*, 2010.

[133] Nick Nikiforakis, Marco Balduzzi, Steven Van Acker, Wouter Joosen, and Davide Balzarotti. Exposing the lack of privacy in file hosting services. In *LEET*, 2011.

[134] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 479–498, 2013.

[135] Aditya Oak, Amir M Ahmadian, Musard Balliu, and Guido Salvaneschi. Enclave-based secure programming with j e. In *2021 IEEE Secure Development Conference (SecDev)*, pages 71–78. IEEE, 2021.

[136] Aditya Oak, Amir M Ahmadian, Musard Balliu, and Guido Salvaneschi. Language support for secure software development with enclaves. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16. IEEE, 2021.

[137] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Cloudav: N-version antivirus in the network cloud. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, 2008.

[138] Jon Oberheide, Kaushik Veeraraghavan, Evan Cooke, Jason Flinn, and Farnam Jahanian. Virtualized in-cloud security services for mobile devices. In *Proceedings of the first workshop on virtualization in mobile computing*, pages 31–35. ACM, 2008.

[139] Hyunyoung Oh, Kevin Nam, Seongil Jeon, Yeongpil Cho, and Yunheung Paek. Meetgo: A trusted execution environment for remote applications on fpga. *IEEE Access*, 9:51313–51324, 2021.

[140] OMTP. Advanced trusted environment: Omtp tr1. `http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf`, 2009.

[141] OASIS Open. Sharing threat intelligence just got a lot easier! `https://oasis-open.github.io/cti-documentation/`.

[142] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253, 2017.

[143] Mike Pall. The LuaJIT project. *Web site: http://luajit.org*, 1015, 2008.

[144] Vasilis Pappas, Vasileios P Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D Keromytis. Cloudfence: Data flow tracking as a cloud service. In *International Workshop on Recent Advances in Intrusion Detection*, pages 411–431. Springer, 2013.

[145] Jaemin Park. Security-enhanced cloud vpn with sgx and enclave migration. 2019.

[146] Sérgio Augusto Gomes Pereira, David Martins Cerdeira, Cristiano António Azevedo Rodrigues, and Sandro Pinto. Providing trusted execution environments using fpga. 2022.

[147] Travis Peters, Reshma Lal, Srikanth Varadarajan, Pradeep Pappachan, and David Kotz. Bastion-sgx: Bluetooth and architectural support for trusted i/o on sgx. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–9, 2018.

[148] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX security symposium*, pages 179–194. San Diego, USA, 2004.

[149] Nick L Petroni Jr and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115, 2007.

[150] Global Platform. Global platform device technology tee system architecture. *Public Review Draft*, 2011.

[151] Global Platform. Globalplatform technology tee system architecture version 1.3. `https://higherlogicdownload.s3.amazonaws.com/GLOBALPLATFORM/transferred-from-WS5/GPD_TEE_SystemArchitecture_v1.3_PublicRelease.pdf`, 2022.

[152] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, 2018.

[153] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM, 2010.

[154] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *EnclaveDB: A Secure Database using SGX*, page 0. IEEE, 2018.

[155] Qualcomm. Guard Your Data with the Qualcomm Snapdragon Mobile Platform. `https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/guard_your_data_with_the_qualcomm_snapdragon_mobile_platform2.pdf`.

[156] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed*. Springer Nature, 2014.

[157] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[158] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. Vcr: App-agnostic recovery of photographic evidence from android device memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 146–157. ACM, 2015.

[159] Samsung. Samsung TEEGRIS. `https://developer.samsung.com/teegris/overview.html`.

[160] Samsung. White Paper : An Overview of Samsung KNOX. `http://www.samsung.com/my/business-images/resource/white-paper/2013/11/Samsung_KNOX_whitepaper_An_Overview_of_Samsung_KNOX-0.pdf`, 2013.

[161] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. ASPLOS, 2014.

[162] Vasily A Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. Eactors: Fast and flexible trusted computing using sgx. In *Proceedings of the 19th International Middleware Conference*, 2018.

[163] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting third party attestation for intel® sgx with intel® data center attestation primitives. *White paper*, 2018.

[164] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: trustworthy data analytics in the cloud using sgx. In *IEEE Symposium on Security and Privacy*, 2015.

[165] Carlos Segarra González. Using trusted execution environments for secure stream processing of medical data. B.S. thesis, Universitat Politècnica de Catalunya, 2019.

[166] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[167] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, 2007.

[168] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. Establishing mutually trusted channels for remote sensing devices with trusted execution environments. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017.

[169] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[170] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. *ACM SIGCOMM Computer Communication Review*, 45(4):213–226, 2015.

[171] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[172] Atsuko Shimizu, Daniel Townley, Mohit Joshi, and Dmitry Ponomarev. Ea-plru: Enclave-aware cache replacement. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2019.

[173] Shweta Shinde, Jinhua Cui, Satyaki Sen, Pinghai Yuan, and Prateek Saxena. Binary compatibility for sgx enclaves. *arXiv preprint arXiv:2009.01144*, 2020.

[174] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*, 2017.

[175] Junliang Shu, Yuanyuan Zhang, Juanru Li, Bodong Li, and Dawu Gu. Why data deletion fails? a study on deletion flaws and data remanence in android systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(2):61, 2017.

[176] Leonardo Humberto Silva, Marco Tulio Valente, and Alexandre Bergel. Refactoring legacy javascript code to use classes: The good, the bad and the ugly. In *Mastering Scale and Complexity in Software Reuse: 16th International Conference on Software Reuse, ICSR 2017, Salvador, Brazil, May 29-31, 2017, Proceedings 16*, pages 155–171. Springer, 2017.

[177] Jagsir Singh and Jaswinder Singh. Detection of malicious software by analyzing the behavioral artifacts using machine learning algorithms. *Information and Software Technology*, 121:106273, 2020.

[178] William Stallings, Lawrie Brown, Michael D Bauer, and Michael Howard. *Computer security: principles and practice*, volume 3. Pearson Upper Saddle River, 2012.

[179] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368, 2003.

[180] Hao Sun, Xiaofeng Wang, Jinshu Su, and Peixin Chen. Rscam: Cloud-based anti-malware via reversible sketch. In *International Conference on Security and Privacy in Communication Systems*, pages 157–174. Springer, 2015.

[181] Kuniyasu Suzaki, Kenta Nakajima, Tsukasa Oi, and Akira Tsukamoto. Ts-perf: General performance measurement of trusted execution environment and rich execution environment on intel sgx, arm trustzone, and risc-v keystone. *IEEE Access*, 9:133520–133530, 2021.

[182] Symantec Enterprise Blogs / Threat Intelligence. The Ransomware Threat Landscape: What to Expect in 2022.

[183] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 665–678, 2018.

[184] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in intel sgx. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 22–27, 2018.

[185] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. Sgxkernel: A library operating system optimized for intel sgx. In *Proceedings of the Computing Frontiers Conference*, pages 35–44. ACM, 2017.

[186] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, page 2. ACM, 2018.

[187] Trustsonic. Kinibi-520a: The latest Trustonic Trusted Execution Environment (TEE). `https://www.trustonic.com/technical-articles/kinibi-520a-the-latest-trusted-execution-environment-tee`.

[188] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[189] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E Porter. Civet: An efficient java partitioning framework for hardware enclaves. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[190] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[191] Marten Van Dijk, Ari Juels, Alina Oprea, Ronald L Rivest, Emil Stefanov, and Nikos Triandopoulos. Hourglass schemes: how to prove that cloud files are encrypted. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 265–280. ACM, 2012.

[192] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Pixelvault: Using gpus for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142, 2014.

[193] Giorgos Vasiliadis and Sotiris Ioannidis. Gravity: a massively parallel antivirus engine. In *International Workshop on Recent Advances in Intrusion Detection*, pages 79–96. Springer, 2010.

[194] Amit Vasudevan, Jonathan M McCune, and James Newsome. *Trustworthy execution on mobile devices*. Springer, 2014.

[195] Hiie Vill. Sgx attestation process, 2017.

[196] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full sql query support. *Proc. Priv. Enhancing Technol.*, 2019(3):370–388, 2019.

[197] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 681–696, 2018.

[198] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *2010 proceedings ieee infocom*, pages 1–9. Ieee, 2010.

[199] Huibo Wang, Erick Bauman, Vishal Karande, Zhiqiang Lin, Yueqiang Cheng, and Yinqian 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*, pages 114–121, 2019.

[200] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2333–2350, 2019.

[201] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *International Workshop on Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[202] Yong Wang, June Li, Siyu Zhao, and Fajiang Yu. Hybridchain: A novel architecture for confidentiality-preserving and performant permissioned blockchain using trusted execution environment. *IEEE access*, 8:190652–190662, 2020.

[203] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. In *Proceedings of the seventh ACM on conference on data and application security and privacy*, pages 261–268, 2017.

[204] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. *ACM SIGARCH Computer Architecture News*, 45(2):81–93, 2017.

[205] Chuliang Weng, Qian Liu, Kenli Li, and Deqing Zou. Cloudmon: monitoring virtual machines in clouds. *IEEE Transactions on Computers*, 65(12):3787–3793, 2016.

[206] Peter Williams and Rick Boivie. Cpu support for secure executables. In *International Conference on Trust and Trustworthy Computing*, pages 172–187. Springer, 2011.

[207] Thu Yein Win, Huaglory Tianfield, and Quentin Mair. Detection of malware and kernel-level rootkits in cloud computing environments. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, pages 295–300. IEEE, 2015.

[208] Ke Xia, Yukui Luo, Xiaolin Xu, and Sheng Wei. Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 301–306. IEEE, 2021.

[209] Jia Xu, Ee-Chien Chang, and Jianying Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 195–206. ACM, 2013.

[210] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors

in android applications. In *European symposium on research in computer security*, pages 163–182. Springer, 2014.

[211] Miao Yu, Virgil D Gligor, and Zongwei Zhou. Trusted display on untrusted commodity platforms. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 989–1003, 2015.

[212] Rui Yuan, Yu-Bin Xia, Hai-Bo Chen, Bin-Yu Zang, and Jan Xie. Shadoweth: Private smart contract on public blockchain. *Journal of Computer Science and Technology*, 33:542–556, 2018.

[213] Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi. Montsalvat: Intel sgx shielding for graalvm native images. In *Proceedings of the 22nd International Middleware Conference*, pages 352–364, 2021.

[214] Peterson Yuhala, Michael Paper, Timothée Zerbib, Pascal Felber, Valerio Schiavoni, and Alain Tchana. Sgx switchless calls made configless. *arXiv preprint arXiv:2305.00763*, 2023.

[215] Denghui Zhang, Guosai Wang, Wei Xu, and Kevin Gao. Sgxpy: Protecting integrity of python applications with intel sgx. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 418–425. IEEE, 2019.

[216] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*, pages 270–282, 2016.

[217] Hanlin Zhang, Yevgeniy Cole, Linqiang Ge, Sixiao Wei, Wei Yu, Chao Lu, Genshe Chen, Dan Shen, Erik Blasch, and Khanh D Pham. Scanme mobile: a cloud-based android malware analysis service. *ACM SIGAPP Applied Computing Review*, 16(1):36–49, 2016.

[218] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. Shef: Shielded enclaves for cloud fpgas. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1070–1085, 2022.

[219] Shixuan Zhao, Mengyuan Li, Yinqian Zhangyz, and Zhiqiang Lin. vsgx: Virtualizing sgx enclaves on amd sev. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 321–336. IEEE, 2022.

[220] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.

[221] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2182–2194, 2021.

[222] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, et al. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1450–1465. IEEE, 2020.

Appendix A
Publications

Related Publications

The research activity related to this thesis has so far produced the following publications (ordered by publication date):

(1) Dimitris Deyannis, Dimitris Karnikis, Giorgos Vasiliadis and Sotiris Ioannidis. **Andromeda: Enabling Secure Enclaves for the Android Ecosystem.** *In Information Security: 24th International Conference (ISC)*. November 2021, Online.

(2) Nikolaos Chalkiadakis, Dimitris Deyannis, Dimitris Karnikis, Giorgos Vasiliadis and Sotiris Ioannidis. **The Million Dollar Handshake: Secure and Attested Communications in the Cloud.** *In 2020 IEEE 13th International Conference on Cloud Computing (IEEE CLOUD)*. October 2020, Online.

(3) Dimitris Deyannis, Dimitris Karnikis, Giorgos Vasiliadis and Sotiris Ioannidis. **An enclave assisted snapshot-based kernel integrity monitor.** *In Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*. April 2020, Online.

(4) Dimitris Deyannis, Eva Papadogiannaki, Giorgos Kalivianakis, Giorgos Vasiliadis and Sotiris Ioannidis. **TrustAV: Practical and Privacy Preserving Malware Analysis in the Cloud.** *In Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY)*. March 2020, Online.

Other Publications

Parts of the research activity performed during this Ph.D. program, not directly related to this thesis, have so far produced the following publications (ordered by publication date):

(1) Dimitris Deyannis, Eva Papadogiannaki, Grigorios Chrysos, Konstantinos Georgopoulos, Sotiris Ioannidis. **The Diversification and Enhancement of an IDS Scheme**

for the Cybersecurity Needs of Modern Supply Chains. *In Electronics 2022.* January 2022.

(2) Eva Papadogiannaki, Dimitris Deyannis and Sotiris Ioannidis. **Head(er)Hunter: Fast Intrusion Detection using Packet Metadata Signatures.** *In 2020 IEEE 25th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. September 2020, Online.

(3) Dimitris Deyannis, Rafail Tsirbas, Giorgos Vasiliadis, Raffaele Montella, Sokol Kosta, and Sotiris Ioannidis. **Enabling GPU-assisted Antivirus Protection on Android Devices through Edge Offloading.** *In Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking (EdgeSys).* June 2018, Munich, Germany.

Appendix B
Acronyms

AAS As a Service

ACPI Advanced Configuration and Power Interface

AEX Asynchronous EXit

AIK Attestation Identity Key

AMT Active Management Technology

ANSI American National Standards Institute

AOSP Android Open Source Project

API Application Programming Interface

APK Android Package Kit

ART Android RunTime

ASCII American Standard Code for Information Interchange

BIOS Basic Input/Output System

CA Certificate Authority

CPU Central Processing Unit

CSV Comma-Separated Values

DFA Deterministic Finite Automaton

DMA Direct Memory Access

DNN Deep Neural Network

DoS Denial of Service

DRAM Dynamic Random Access Memory

DRoT Dynamic Root of Trust

EDL Enclave Definition Language

EDMM Enclave Dynamic Memory Management

EFFS Embedded Flash File System

EK Endorsement Key

EL Exception Level

eMMC embedded Multi-Media Card

EPC Enclave Page Cache

EPID Enhanced Privacy Identifier

ERET Exception RETurn

ES6 ECMAScript 6

FEK File Encryption Key

FIRQ Fast Interrupt ReQuest

FLC Flexible Launch Control

FPGA Field Programmable Gate Array

GbE Gigabit Ethernet

GDPR General Data Protection Regulation

GPGPU General Purpose Graphics Processing Unit

GPU Graphics Processing Unit

HAL Hardware Abstraction Layer

HECI Host Embedded Communication Interface

HMAC Hash-based Message Authentication Code

HTTP HyperText Transfer Protocol

IAS Intel Attestation Service

IoT Internet of Things

IPC Inter Process Communication

IRQ Interrupt ReQest

ISA Instruction Set Architecture

IV Initialization Vector

JIT Just-In-Time

JNI Java Native Interface

JS JavaScript

JSON JavaScript Object Notation

JVM Java Virtual Machine

LKM Loadable Kernel Module

LLC Last-Level Cache

LoC Lines of Code

LRU Least Recently Used

MAC Message Authentication Code

ME Management Engine

MEE Memory Encryption Engine

MMIO Memory Mapped I/O

MMU Memory Management Unit

MPU Memory Protection Unit

MRU Most Recently Used

MSR Model Specific Register

NDK Native Development Kit

NFV Network Functions Virtualization

NIC Network Interface Card

NIDS Network Intrusion Detection System

NLP Natural Language Processing

OOM Out Of Memory

NS-bit Non-Secure bit

NSTID Non-Secure TLB ID

OS Operating System

OTP One Time Password

PCI Peripheral Component Interconnect

PCR Platform Configuration Register

PKI Public Key Infrastructure

PMA Protected Module Architecture

PPID Platform Provisioning Identifier

PSP Platform Security Processor

PSW Platform SoftWare

PTBR Page Table Base Register

PTE Page Table Entry

PTR Private and authenticated Tamper-Resistant environment

PTT Platform Trust Technology

QE Quoting Enclave

RA Remote Attestation

RAM Random Access Memory

REE Rich Execution Environment

ROM Read-Only Memory

RoT Root of Trust

RPMB Replay Protected Memory Block

SCR Secure Configuration Register

SDK Software Development Kit

SDN Software Defined Network

SE Secure Executable

SECS SGX Enclave Control Sequence

SEE Secure Execution Environment

SEV Secure Encrypted Virtualization

SGX Software Guard eXtensions

SK Security Kernel

SMC Secure Monitor Call

SME Secure Memory Encryption

SMI System Management Interrupt

SMM System Management Mode

SMRAM System Management RAM

SNP Secure Nested Paging

SP Service Provider

SPI Serial Peripheral Interface

SPID Service Provider Identifier

SRAM Static Random Access Memory

SSK Secure Storage Key

TA Trusted Application

TCB Trusted Computing Base

TEE Trusted Execution Environment

TEEOD Trusted Execution Environment On-Demand

TLB Translation Lookaside Buffer

TLS Transport Layer Security

TPM Trusted Platform Module

TSK TA Storage Key

TSME Transparent Secure Memory Encryption

TTL Time-To-Live

TXT Trustex eXecution Technology

UEFI Unified Extensible Firmware Interface

UUID Universal Unique IDentifier

VFS Virtual File System

VMM Virtual Machine Manager