Partitioning the Trusted Computing Base of Applications on Commodity Systems

Ahmad Atamli

University College

University of Oxford

A dissertation submitted for the degree of

Doctor of Philosophy

Trinity 2017

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Prof. Andrew Martin, for his insight, guidance, and support throughout my DPhil. I am very grateful for all the collaborative research effort and work, freedom, and for the many opportunities he has provided. I also appreciate the various interactions I have had with Ivan Martinovic, Ivan Flechais, Kasper Rasmussen, Andrew Simpson, and David Grawrock during my time in the Department of Computer Science. I thank my colleagues, especially Ranj, Nalin, Andrew, Ravi, Pardeep, Robin, Anbang, Yudhi, Nikola, and all the members of the Cyber Security Centre, for the many stimulating discussions we shared. I thank my former Professors at Technion, especially Isaac Keslassy and Michael Cwikel for their support and inspiration to do the foundations for this research endeavour. I thank my team and the entire Mellanox Technologies family, especially Peter Paneah and Ofir Arkin for their endless support and encouragement.

I am grateful to University College for adding to my Oxford experience, and especially to Nikola, Leo, Thibo, Kusha, Lucie, Hannah, Jonathan, Owen, and Inbar for their support and friendship. In particular, I thank my mates at the Hall of Brohan and OUPLC for the good time training, food, nights out, and massive support in the last three years. I will never forget the good times with Jonathan, Dom, Scott, Matthew, Jamie, Zander and the rest of OUPLC. In particular, I thank my dear friend Pistaki for all the laughs, joint training sessions, and nights out together.

I thank my dear friends Giuseppe Petracca and Kusha Baharlou- the inventor of the Dr. Jack Hammer and Mr. Fasolye, for the support, for listening, and for countless adventures and life-time experiences. I also thank Fishi for the experiences, understanding, candy, milki, food, inspiration, and joyful moments.

I have had the privilege of meeting many amazing people during my time in Oxford and so I thank all my friends from the Department of Computer Science, University College, the OUPLC, and Hall of Brohan for all the memorable experiences.

For my family, Granny, Yazeed, Mum, and Dad. It has been a long ride, and I thank you for being always there for me. For your patience, tolerance, love, unfailing encouragement, and understanding. It would have not been the same without you.

Abstract

Secure containers implemented in both software and hardware are being used to isolate and reduce attack vectors on executing software. The isolation of software partitions protects the data and the execution from external software (e.g. the OS, other applications, other software partitions within the same application). Despite the existence of many hardware isolation technologies such as ARM TrustZone, Intel Software Guard Extension (SGX), and others, it is still not clear how to efficiently use isolation to secure applications data. This is particularly the case when considering vulnerabilities within the application, strong adversaries who have control over the OS, and performance requirements of the application.

Previous work demonstrated the efficiency of SGX in protecting against memory leakage vulnerabilities. However, since SGX allows separation of privileges through partitioning monolithic applications into compartments, using it in mitigating faulty API vulnerabilities or Buffer over-writes is far from being straightforward. To illustrate, we found that many systems with "secure containers" capabilities do not deliver the security expected from containers, which indicates an absence of a methodology for using Trusted Execution Environment (TEE) systems. For example, our analysis of Samsung KNOX architecture revealed that such systems cannot protect against memory leakage, buffer over-reads, buffer over-writes, and others.

In this thesis two research hypotheses are investigated. First, privilege separation through application partitioning enhanced TEE can be used to mitigate software vulnerabilities, protect containers from privileged kernel, while maintaining the reasonable performance of an application . Second, partitioning patterns can be used to mitigate different threats. We demonstrate how vulnerabilities can be mitigated with secure containers and how the specific design of the secure containers determines the success of the desired protection from such a paradigm.

This research uncovers the potential of TEE in separating privileges in applications using hardware based technologies instead of access control enforced in several layers by software. The realisation of the potential of secure containers will help corporations and enterprises design better secure systems and devices that protect end-users data from application and system vulnerabilities and attacks performed by software.

Publications

The following publications and presentations have resulted from this research as well as others. Conference and workshop papers have been published or presented in formal proceedings. Other papers include research undertaken with collaborators where my specific contribution is listed, or research that was not included as part of this dissertation. Any text from the publications which is used in this thesis is the product of my independent work.

Conference/Workshop/Journal Papers

- A. **Atamli** Reineh, A. Paverd, G. Petracca, and A. Martin, "A framework for application partitioning using trusted execution environments," *Concurrency and Computation: Practice and Experience*, pp. 1–23, 2017

- A. **Atamli** Reineh, R. Borgaonkar, R. A. Balisane, G. Petracca, and A. Martin, "Analysis of trusted execution environment usage in samsung knox," in *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, pp. 7:1–7:6, ACM, 2016

- A. **Atamli** Reineh, G. Petracca, J. Uusilehto, and A. Martin, "Enabling secure and usable mobile application: Revealing the nuts and bolts of software tpm in todays mobile devices," *arXiv preprint arXiv:1606.02995*, 2016

- A. **Atamli** Reineh and A. Martin, "Securing application with software partitioning: A case study using sgx," in *Proceedings of 11th International Conference on Security and Privacy in Communication Systems, SecureComm*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pp. 605–621, Springer, 2015

- A. W. **Atamli** and A. Martin, "Threat-Based Security Analysis for the Internet of Things," in *Proceedings of the 2014 International Workshop on Secure Internet of Things*, SIOT '14, (Washington, DC, USA), pp. 35–43, IEEE Computer Society, 2014

Other Papers:

- G. Petracca, A. **Atamli** Reineh, Y. Sun, J. Grossklags, and T. Jaeger, "Aware: Preventing abuse of privacy-sensitive sensors via operation bindings," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 379–396, USENIX Association, 2017

  - My contribution is in the binary analysis of the application for tables in section 8 and 9, and the text in defining the attacks scenarios, and related work.

- J. R. Nurse, A. **Atamli**, and A. Martin, "Towards a usable framework for modelling security and privacy risks in the smart home," in *International Conference on Human Aspects of Information Security, Privacy, and Trust*, pp. 255–267, Springer, 2016

  - My contribution is defining the use-cases and security risks of each use-case.

- G. Petracca, Y. Sun, T. Jaeger, and A. **Atamli**, "Audroid: Preventing attacks on audio channels in mobile devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*, pp. 181–190, ACM, 2015

  - My contribution is in defining part of the trust model in section 3, and some of the attacks scenarios and their relevant part in the implementation.

- A. **Atamli** Reineh, A. J. Paverd, and A. P. Martin, "Trustworthy and secure service-oriented architecture for the internet of things," *arXiv preprint arXiv:1606.01671*, 2016

  - This paper has been presented as a poster in the International Conference of the Internet of Things 2014 (Boston, USA).

List of Abbreviations

AES	Advanced Encryption Standard
AIK	Attestation Idendity Key
CA	Certification Authority
CFG	Control Flow Graph
DIFC	Decentralised Information Flow Control
DRTM	Dynamic Root of Trust for Measurement
DTA	Dynamic Taint Analysis
EPC	Enclave Page Cache
EPCM	Enclave Page Cache Map
FSM	Finate-State-Machine
GCM	Galois/Counter Mode
ICE	Isolated Computing Environment
ISR	Instruction-set randomisation
IV	initialisation Vector
JVM	Java Virtual Machine
LKM	Linux Kernel Module
LLMV	Low Level Virtual Machine
LMA	Local Mobile Application
LOC	Lines of Code
MAC	Message Authentication Code
MMU	Memory Management Unit
OEM	Original Equipment Manufacturer
OS	Operating System
PAL	Piece of Application Logic
PCR	Platform Configuration Registers
RFC	Internet Security Glossary
SD	Standard Deviation

SDK	Software Development Kit
SDN	Software Defined Network
SGX	Software Guard Extension
SK	Storage Key
SOAAP	Security-Oriented Analysis of Application Programs
SSL	Secure Sockets Layer
TBS	TPM Base Services
TC	Trusted Computing
TCB	Trusted Code Base
TCG	Trusted Computing Group
TDC	Trusted Domain Controller
TEE	Trusted Execution Environment
TIMA	TrustZone-based Integrity Measurement Architecture
TLS	Transport Layer Security
TPM	Trusted Platform Module
TXT	Trusted Execution Technology
VM	Virtual Machine
VMM	Virtual Machine Monitor
XSS	Cross-Site Scripting

Table of Contents

List of Figures

List of Tables

I think computer viruses should count as life. I think it says something about human nature that the only form of life we have created so far is purely destructive. We've created life in our own image.

Stephen Hawking

1

Introduction

Contents

The world is experiencing rapid advancement in technology, which has had a significant impact on our daily lives. Companies reduce the cost of maintenance and operations through automation, consumers extensively use cloud services, and researchers improve their global learning using global networks. Furthermore, the need to meet consumer demands of functionality, scalability, and automation generates more complex and larger applications.

The increase in numbers of lines of code in applications persists, and so do the numbers of software vulnerabilities. Thus, exploits continue to cause significant damage, especially leaking sensitive data. Despite this, applications and operating systems remain monolithically structured, and all the functionalities are given the same privileges whether or not those functionalities are sensitive to the overall security of the application. As stated by others [10, 11, 12, 13, 14] the shortcoming is due to the limited and coarse isolation primitives the Operating System (OS) offers.

Fortunately, Intel Software Guard Extension (SGX) and ARM Trustzone are isolation primitives that are rooted in hardware. The hardware primitive referred to as Trusted Execution Environment (TEE) is tamper-resistant and reports on the state of the software inside that TEE. If users trust the manufacturers of the hardware and the design of the platform, then they can believe in the isolation provided and its reporting on the software. In theory, software assurance methods such as testing and verification should be able to point out vulnerabilities in software. However, these methods are not absolute and attacks during runtime are still possible. There appear to be significant practical problems in mitigating software vulnerabilities during run-

time. These methods primarily limit the impact of an exploit of vulnerability and whether the software that runs on a platform protects the confidentiality of sensitive data.

However, even with the presence of less coarse isolation primitives, it is still unclear how to partition applications into secure compartments [15, 16], and how to secure information flowing from the user to prevent leaking information in every step of the execution [17].

This thesis addresses the problem of partitioning for securing confidentiality of data in applications, and answers the following questions:

- What are the benefits of application partitioning into isolated compartments?
- What are the issues faced when splitting applications, and how significant are they?
- What is missing to bring the realisation of such approach to design secure applications?
- To what extent is application partitioning practical and how does it affect application execution?

The following sections begin with an introduction to the shortcomings of the monolithic structure adopted by many applications. It then discusses how previous research has aimed to solve this issue by splitting and separating privileges. Next, it introduces the security methods used to enforce separation between the partitions. Then

the working hypothesis behind this research is presented along with the methodology adopted. Finally, the structure and contributions of this dissertation are presented.

1.1 Monolithic Applications

What is a *monolithic application*? In software engineering [18] *a monolithic application describes a single-tiered software application in which the user interface and data access code are combined into a single program from a single platform.* In technical terms, a monolithic application is an application that adopts a monolithic structure, in which data and code run in one process under the same privileges. To clarify, a monolithic C/C++ application follows a monolithic structure of process memory in which any code has access to the entire data plane of a process memory, heap, and stack. The aforementioned allows untrustworthy code such as a third-party library to tamper with the execution.

Despite the wide adoption of the monolithic structure in designing applications for many years, it can be argued that a monolithic application is fragile, unreliable, and is bad for application security [19, 20]. It has limitations in several aspects of security including the inability of segmentation between the different users [21, 22], mitigating software vulnerabilities [23, 24], and isolating sensitive data flows [25, 26]. However, this topic is controversial, some researchers believe that the issue of monolithic applications is a matter of software engineering, in which the protection is statica or dynamic using compilers. Others [27, 28, 29, 30, 31] believe the issue of software

engineering should be addressed by better advancing the research on programming languages and how the pre-run and run-time stages can enhance the security of monolithic applications. These approaches assume a weak adversary and strong trust model where the isolation performed by the OS is trusted. Other researchers believe that while it's possible it remains hard to perform successful attacks [32, 33]. Nonetheless, this thesis assumes strong adversary that aims to tamper with the execution of sensitive components resulting in the leakage of sensitive data. Thus, the rationale is to adopt non-monolithic structure that relies on hardware to enforce the isolation.

In an attempt to address the security-related shortcomings in monolithic applications, the research space of splitting applications became an essential realm in the security community. This area is also referred to as *privilege separation* in applications which is based on splitting monolithic applications into several partitions with different privileges enforced by different primitives of hardware and software. The partitions are known by names such as enclaves [34], containers [35], and rendering engines [36]. Notwithstanding, they all refer to the isolation of one partition from external software. This thesis uses the word *containers* when referring to the partitions when splitting applications. However, for the sake of easier comprehension, technology-specific terminology such as Intel SGX enclave shall be retained in their original form. Furthermore, the containers used henceforth are based on hardware primitives and are referred to as TEE.

The reason for splitting an application into several compartments is to enhance its security. Previous research used privilege separation to prevent untrusted blocks

from accessing privilege operations [37, 38], and privileged separation has also been used to prevent the flow of sensitive operations to untrusted software. However, these approaches assume strong trust of the OS and do not consider low level system vulnerabilities involving the memory.

Vulnerabilities have appeared in many systems in the last decade. For instance, buffer overruns have been a known security problem for a long time[39]. In fact, the problem of buffer over-runs go back as far as the 1960. Exploitation of buffer over-run vulnerabilities has a sever impact, including privilege escalation, memory corruption, and random jumps among others. According to Microsoft Response Centre [40] the estimated cost of issuing one security announcement on a vulnerability, patching the error, and system administrator working hours costs hundreds of thousands of dollars. This estimation doesn't factor in the impact incurred by a vulnerable system until a fix is in place.

There are many different reasons for vulnerabilities in systems. One of the main and obvious reasons for vulnerabilities in systems is the size of the Trusted Computing Base (Trusted Code Base (TCB)). The likelihood for vulnerabilities in code increases linearly with the growth of the number of Lines of Code (LOC). The larger the LOC is the higher the likelihood for errors and flaws in the software design and code[41, 42]. Despite the improvement in high-level languages such as Java/C# and their run-time checking of array boundaries (which enhance vulnerability detection and prevention), it is still the case that C and C++ are used to write operating systems and many other applications due to their superior flexibility, power and speed. Hence, there is

a need to introduce other approaches to mitigate those vulnerabilities.

In fact, many protection techniques were developed to detect faults in code or to prevent attacks [43, 44, 45, 46, 47]. However, these techniques, rely on a set of known vulnerability patterns to identify faults in code. In the presence of a new unknown vulnerability pattern in the code such methods fail. Studies show that those techniques are imprecise and rely heavily on pre-analysis of the code and deploying protection manually [48].

Applications have long since surpassed the feasible limit for assurance techniques such as formal verification to verify the correctness of the code, and numerous factors have rendered manual review equally insufficient for that task. Accompanying the growth of the code in Cloud applications, more classes of attacks have been identified [49], such as stealing secrets and modifying sensitive code [50, 51]. These attacks either exploit vulnerabilities in (Virtual Machine Monitor (VMM)) or in applications directly [52, 53, 54]. A typical cloud system is very complex and has to address many issues concerned with access. A cloud system must facilitate access to the network, storage, application, and Virtual Machine (VM) among others, for many users, making it very complex and, relatedly, vulnerable.

One of the primary sources of vulnerabilities in applications is the imported third party libraries. Applications grew massively in the number of LOC, and the more functionalities an application provides the more LOC it has. Many application developers tend to use third-party libraries to reduce the time of the development process.

Also, applications have many components and are being developed by several developers. Unfortunately, security is developed poorly in components of an application and a vulnerability in as little as one part of the application could lead to information leakage and cause fatal losses to enterprises and cooperations. Several examples of vulnerabilities that caused losses were found in well-known libraries and parts of systems such as OpenSSL [24], eCryptFS [55], fingerprint authentication code [56, 57], Certificate verification code [58, 59, 60] and others [55].

1.2 Isolation between Partitions

Modern operating systems and applications are growing at a rapid pace, and carry numerous security vulnerabilities and increasing the attack vectors that accompany such growth. According to Robles et al. [61] operating systems such as Linux, Windows, Android, and iOS have grown massively over the last decade. In addition, the increase of functionalities in applications has resulted in larger code and hence larger environment space of processes. The most common approach for isolation is to use privileged code such as an OS or VMM that enforces access control semantics [62, 63, 64, 65, 66]. A VMM will typically use hardware assistance for virtualisation, however, the access control is enforced by software using meta-data of a memory address table. In contemporary operating systems the OS enforces access control between processes. Each process has its own code and data in memory, and the OS prevents one process from accessing another process' space, including memory

addresses and executable code. Modern operating systems such as Android use the kernel to provide isolation between applications. The android system sets up kernel-level application sandbox and assigns a unique user id to each Android application and runs it as that user in a separate process. The kernel enforces security between applications and the system at the process level through standard Linux facilities, such as user and group IDs that are assigned to applications.

Similarly, isolation can be achieved in hardware. ARM TrustZone [67] and Intel SGX [68] provide TEE capability, which isolates software partition from the rest of the system in hardware. For instance, Intel SGX enables instantiating a TEE (called enclave) which encrypts enclave data in memory and protects the enclave from external software. In addition, access to the enclave is achieved through a pre-defined method only. In the case of ARM TrustZone, the isolation is achieved by physically splitting the memory and CPU caches between the secure world and normal world.

1.3 Privilege Separation

Previous sections listed methods for protecting partitions through enforcing isolation between the different partitions. However, splitting applications into partitions is a formidable task. In many operating systems e.g. Linux and Android, applications run under a process and the OS enforces access between the processes. For the most part, every application has its own process and cannot be accessed by another application. In Linux, every process has its process environment and is tied to a user ID. The

OS authorises access to the application's process space and prevents unauthorised access of other applications. It is important to note that an application developer can separate his application in two different processes[69, 38, 70, 71], however, it is the responsibility of the developer to establish a communication channel between processes of the same application if they wish to access the data/code of each other.

Another way used to protect software partitions is the least privilege principle. The principle of least privilege [72] involves dividing the code into compartments, each of which executes with the minimum privileges needed to complete its task. Such an approach not only limits the harm malicious injected code may cause, but can also prevent bugs from accidentally leaking sensitive information.

Programmers frequently have a good idea which data manipulated by their code is sensitive, and a similarly good idea which code is most risky (typically because it handles user input). *So why do so few programmers divide their code into various privileged compartments?* As others have noted [51, 73], one reason is that the isolation primitives provided by today's operating systems grant privileges by default, and so are cumbersome when used to limit privilege.

Consider the use of processes as compartments, and the behaviour of the fork system call: by default a child process inherits a clone of its parent's memory, including any sensitive information therein. To prevent such implicit granting of privilege to a child process, the parent can scrub all sensitive data from memory explicitly. But doing so is brittle; if the programmer neglects to scrub even a single piece of sensitive

data in the parent, the child gains undesired read privileges. Moreover, the programmer may not even know of all sensitive data in a process' memory; library calls may leave behind sensitive intermediate results.

1.4 Research Hypotheses

As discussed earlier in this chapter, monolithic applications suffer from (1) absence of separation of privileges of application functionality, (2) software vulnerabilities due to the large size of the code, and (3) poor isolation of sensitive information flows. Thus, it is essential when redesigning the security of an application to keep those three shortcomings in mind. In addition, it is essential to investigate the shortfalls and merits of the new design and its propagation on security research such as formal verification, software development, and vulnerabilities exploit. Having identified the main security challenges of securing information flow and privileged operation from system capable adversary, this thesis proposes a solution based on containers enhanced TEE and describes the investigation of two primary research hypotheses:

1. In applications, TEE can be used to enhance overall security and mitigate software vulnerabilities in complex and large scale applications while maintaining the primary functionality and performance requirements of the application. In particular, when must application structre be broken into several parts to allow mitigation of software vulnerabilities.

2. Partitioning patterns can be used to mitigate vulnerabilities and limit the threats of the attack vectors for an application. Under which conditions and patterns this statement holds and what are the drawbacks when using hardware extension.

1.5 Experimental Work/Methodologies

Securing sensitive information from software vulnerabilities is a well known problem that has attracted research for many years [37, 1, 69, 70]. In this research we chose to take a new approach to mitigate systems' vulnerabilities by using hardware extensions that enables running software in a TEE. Choosing to partition applications and securing each partition using hardware extensions to guarantee isolation between the different partitions, thus, preventing an exploit in one partition from obtaining confidential data from other partitions. As discussed earlier, most software solutions increase the size of the TCB and the isolation measures between partitions are not guaranteed when a strong adversary is present and able to compromise the kernel. While currently available hardware technologies that provide isolation make it much harder to compromise the system, even with the presence of a strong adversary. Specifically, the secure world of ARM TrustZone is physically separated from the untrusted normal world and has different environment. By the same token, Intel SGX encrypts data in memory when it leaves the CPU, making it cryptographically impossible to deduce the data in reasonable time. In addition, Intel SGX enclave

provides checks on the data passed to the trusted region to prevent exploiting vulnerabilities. For the aforementioned reasons, isolation using hardware is more robust and can deter a strong adversary.

In order to research the potential of TEE in securing applications, we started by investigating fatal and well known vulnerabilities in applications e.g. Heartbleed in Apache web services [24]. In large applications such as Apache, vulnerabilities are inevitable when the number of LOC is in the range of tens and hundreds of thousands. Splitting Apache into several partitions that are isolated from each other helped us identify which method to apply to mitigate vulnerabilities. Splitting an application into partitions will limit the impact of a vulnerability in one partition on the others. In addition, further study into the Heartbleed vulnerability showed that there is an absence of privilege separation between operations in the OpenSSL library. This was another reason why the mentioned vulnerability has a high impact on applications. The coming chapters will describe further the motivation behind this approach.

Chapter 6 will show how this approach mitigates vulnerabilities in Apache which suffers from multiple vulnerabilities running in one process (one partition) under Linux. To achieve the aforementioned, we applied software partitioning and chose to use secure containers to isolate each partition. We examined how to use software partitioning with secure containers in order to reduce attack vectors and mitigate software vulnerabilities. Hence, we investigated and used SGX as a potential technology providing secure containers (referred to by the SGX documentation as secure enclave) to secure software partitions. Following our investigation with SGX we found that

through suitable methodology for software partitioning, information leakage exploits resulting from buffer over-read can be mitigated. Furthermore, we were able to spot some of the integrity faults resulting from buffer over-write. Our proposed solution in chapter 5 [4] using Intel SGX protected the confidentiality of data leaked by an exploit of a buffer over-read vulnerability. However, for the buffer over-write exploits we could only point to the integrity issue resulting from the over-write attack.

1.6 Contribution and Thesis Structure

This thesis explores the topic of privilege separation through application partitioning. The main contribution of this thesis is building on hardware primitives offering high security guarantees for application partitioning. This research shows the untapped potential of application partitioning when hardware technologies such as ARM TrustZone and Intel SGX are used. In addition, this thesis explores the gaps in previous research on partitioning and partitioning patterns in several case-studies.

The thesis is divided into eight sections and the relationship between them is described in figure 1.1 . This first chapter gave the motivation behind the research and the contributions of this dissertation and set the ground for the following chapters. Chapter 2 gives the background on trusted computing, isolation and software vulnerabilities. It also gives the background to hardware isolation methods. In chapter 3 a thorough gap analysis is conducted to discuss the current state of the research on privilege separation. This chapter discusses the scarce research in the field and

highlights the lack of approaches to application partitioning, as well as discussing the security of these approaches. Chapter 5 presents how to secure authentication protocol in a split of two worlds; trusted and untrusted, using merely two partitions. The complexity is taken one step further in chapter 4 with multiple partitioning schemes. This chapter presents how to mitigate software vulnerabilities using TEEs with several real-world case studies including OpenSSL and KNOX. The approach presented in chapter 4 is further developed in chapter 6, where it is evaluated further by examining the method used against all previously known vulnerabilities in three applications: Apache, OpenSSL, and SQLite. This chapter also demonstrates the practicality of the suggested solution by evaluating the overhead cost incurred. Finally, chapter 8 concludes this dissertation and presents future avenues for continuing this investigation.

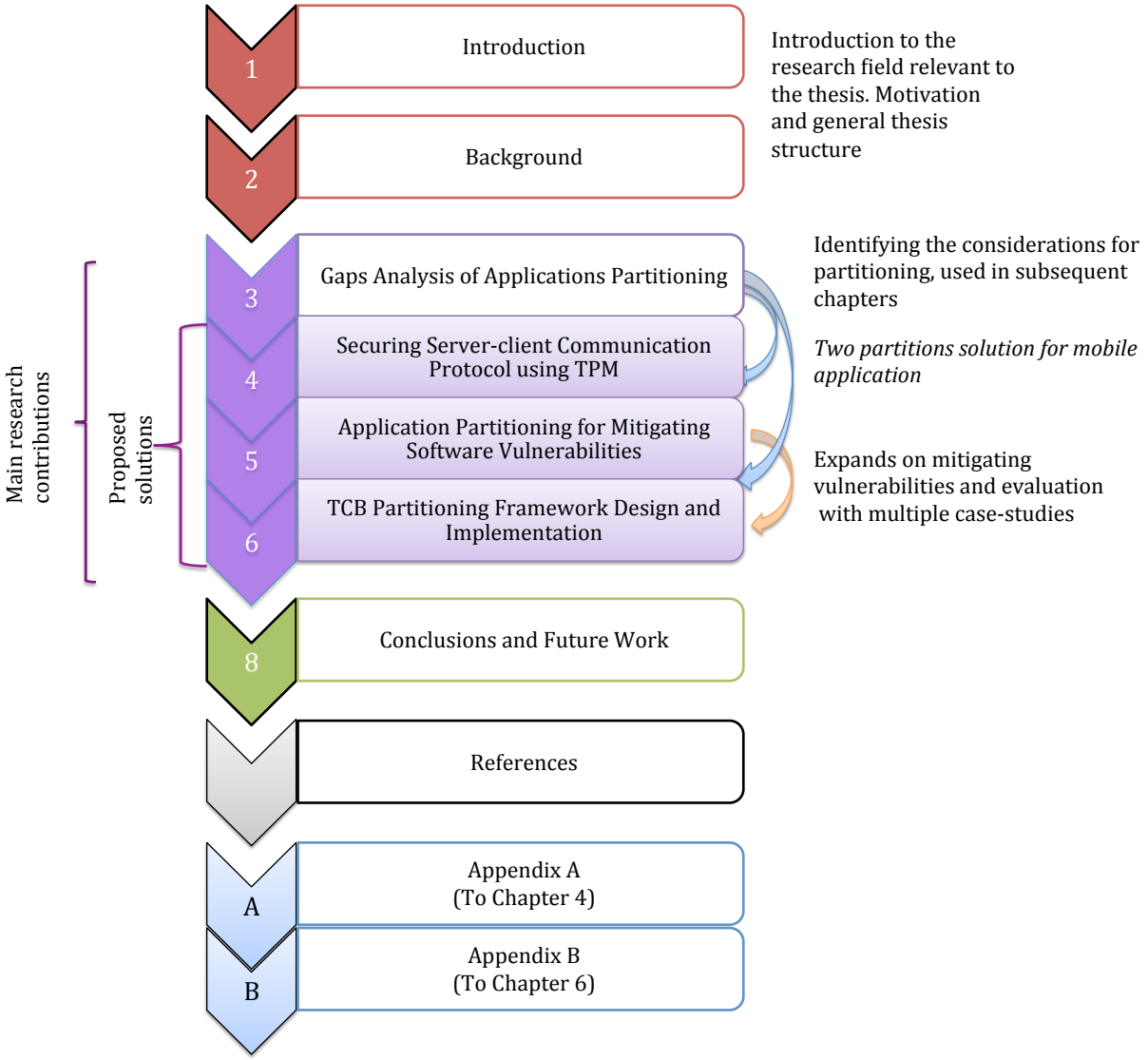

Figure 1.1: Thesis Structure: A list of the main thesis chapters, their content and relationships between them are summarised.

To elaborate further on the specific contributions presented in this dissertation:

- We shed the light on the disperse approaches taken in previous research on application partitioning and suggest that inadequate research on application partitioning in security has been undertaken;
- We present a new client-server trusted authentication protocol based on trusted computing and describe how to secure the proposed protocol by splitting the application into trusted part running in the secure world of the TrustZone and untrused part running under the rich OS;
- We investigate, propose, and implement a partitioning scheme that prevents the leakage of sensitive information when exploiting software vulnerability in OpenSSL and that mitigates a well known high impact software vulnerability: the Heartbleed;
- We propose a framework that systematises application partitioning schemes for TEEs into four application-independent *types*;
- We propose and investigate evaluation criteria for partitioning schemes, and we use these to establish application-independent relationships between the different types of partitioning schemes;
- We explain these partitioning schemes and the relationships between them using three case-studies, in which we evaluate the security properties of each type of scheme using published vulnerabilities.

- We categorise previous work on privilege separation and application partitioning schemes using our new framework.

Although this report contains no information not available in a well stocked technical library or not known to computer experts, and although there is little or nothing in it directly attributable to classified sources, the participation of representatives from government agencies in its preparation makes the information assume an official character. It will tend to be viewed as an authoritative Department of Defence product, and suggestive of the policies and guidelines that will eventually have to be established. As a prudent step to control dissemination, it is classified CONFIDENTIAL overall

Willis Ware,1970 Task Force on Computer Security

2

Background: Trusted Computing, Isolation, Software Vulnerabilities

Contents

This chapter provides background information on the themes relevant to this thesis. Section 2.1 provides background on the concepts of trust and trustworthiness and is followed by explanations of current mechanisms that enable establishing trust in software running under computer systems, including but not limited to Trusted Computing (TC) (Section 2.2) and Trusted Execution Environment (TEE) (Section 2.4).

2.1 Trust and Trustworthiness in Systems

Trust has many meanings, thus, leading to confusion and provoking mistrust when not considering the context it's used for. Since these concepts are the central backbone for this thesis, this section provides background on trust and trustworthiness in systems and the meaning in this context. Trust as defined by Trusted Computing Group (TCG): "An entity can be trusted if it always behaves in the expected manner for the intended purpose" [74]. The trusted entity referred to by TCG can be a computing process, human or anything. The definition does not indicate the *goodness* of what is a trusted entity, it is rather about the belief that an entity will behave in the same manner. To illustrate this in canonical form: *"Alice trusts Bob to do X"* or *"if asked to wash the dishes, Alice trusts Bob to wash the dishes every time"*. When the concept of trust is applied to computational systems, it is manifested on the belief that a system exhibits specific characters and expected behaviour. The Internet Security Glossary (RFC) 4949 [75] defined a trusted computer system as *"A system that operates as*

expected, according to design and policy, doing what is required despite environmental disruption, human user and operator errors, and attacks by hostile parties, and not doing other things". The definition clearly states that a computer system should behave in expected manner despite environmental disruption, however, it has been shown [49, 76, 77, 78] that it is correlated to the size of the Trusted Code Base (TCB).

The Orange Book defines TCB as "the totality of protection mechanisms within it, including hardware, firmware, and software, the combination of which is responsible for enforcing a computer security policy" [79] in which the apex anchor, also referred to as the root of trust, is rooted in hardware as defined by RFC 4949: [75] *"The trust anchor that is superior to all other trust anchors in a particular system or context"*. The anchor is the first element that runs on a computer system and verifies the assumption of the second element expected behaviour, thus, the second element is a trusted system.

Many confuse between the concept of **Trusted System** and **Trustworthy System**: *"A Trustworthy system is not only trusted, but also warrants that trust because the systems behaviour can be validated in some convincing way, such as through formal analysis or code review"* such definition indicates that a *Trustworthy System* has more than trusted expectation from a system. In turn, trusted and trustworthy systems are important for making a *Trust Decision*, if one can confirm the same entity in operation and its properties, the relying party can make a trust decision regarding the entity. The *Trust Decision* is based on the belief that, with the knowledge of the entity's identity, it can be determined by seeking party whether the entity can

be trusted to behave in expected manner and deliver expected services. In essence, trusted components in a system as stated by the Martin et al [80]:

- Strongly identify themselves to other parties.
- Strongly identify their running software and status.
- Allow us to make informed decision about the level of trust to invest in them.

As defined by the Orange Book on Trusted Computing Base, these conditions can be achieved in combination of software, hardware, firmware and there are various methods to guarantee these conditions[81, 82]. This thesis will focus on trusted computing and its hardware primitives as defined in the following sections.

2.2 Trusted Computing

Trusted Computing (TC) is a batch of proposals and ideas combining various technologies and PC architecture which can give guarantees regarding the software it's running, which allows computer systems to communicate securely with each other. TC makes use of hardware support in order to enable the use of security primitives of hardware encryption keys, memory curtaining, secure execution, and tamper resistant. The TCG developed and standardised many TC technologies, the most well-known one is the Trusted Platform Module (TPM). Other hardware-based security technologies developed by companies such as Intel, ARM, Trustonic are also

considered TC technologies. Trusted Execution rooted in hardware is developed both by Intel and ARM and is one of the corner stones of this thesis. This section provides a brief overview of the relevant aspects of TC.

2.2.1 Remote Attestation

Attestation is the process of validating the integrity of all the softwares that have been executed in a computing system. In most systems, the aforementioned software resides within the same TCB. Consequently, *Remote Attestation* refers to the process of communicating the measurement from the attesting entity to the remote entity with protected measurement (also called protected integrity record) of the executing software in the same TCB at the attesting entity platform. In many systems, this information is used by the remote entity to verify the integrity of the attesting entity, thus, enabling the remote entity to make a decision on the trustworthiness of the attesting entity. As a result of the verification process, the remote entity may consider the attesting entity as trusted.

Unfortunately, the measurement of all executables in a traditional platform includes many executable binaries, thus, changes in any of the executables lead to different measurement and as a result require updated record at the remote entity. Lyle and Martin quantified the measurement from a web server and client PC and concluded that frequency of the change of the measurement are far more frequent than anticipated [83]. Their results recorded 1137 changes to the measurements over

a 32 month period, making it hard for the verfying entity to keep track with allowed records.

In the cloud environment, where services are provided for many users, it is impractical for remote users to keep up with every change in the web server. It is more practical for a remote user to verify smaller number of entities, for instance, the application its communicating with. However, if other modules in the same TCB are compromised, the measurement of specific software modules might not be trustworthy, since the presence of unreported malware in the system defeat the purpose of the security aspect of remote attestation. On the othe hand, in the presence of multi-TEE system, multiple TCB can run on the same system. For instance, an application such as Apache can run in TEE, hence, the TCB is limited to the Apache code which is much smaller in size than full software stack of a web server.

In addition, in modern systems remote attestation is not limited to the code. The remote entity may require from the attesting entity to attest other characters such as control flow of execution, timing of execution, and server configurations. The next section describes the technology that enables fine-grained remote attestation and *how* a web server can prove to remote entity the execution in separate TEE.

2.3 Isolation

The term isolation describes the isolation mechanisms focusing on protecting an environment from access by untrusted entities. The environment being protected by isolation mechanisms is referred to by many names such as container [35], chambers [3], and enclaves [62]. In this thesis, **container** is used to describe the protected environment. It is the responsibility of an isolation mechanism to isolate between the different containers running under a computer system and protect the containers from higher privileged software such as the Operating System (OS).

There are different patterns for containers, the most well-known patterns are sandboxing and sensitive work compartment. *Sandboxing compartment* aims to isolate untrusted entity inside a container from the rest of the system. The trust relationship between the system and the sandbox container is that the system cannot trust the code running inside the container. For instance, the code might have vulnerabilities that affect the rest of the system. The other infamous type is *sensitive work compartment*, this pattern aims to protect the execution and the data in-transit produced in the container. The confidentiality and integrity of the compartment data and the tenant software are protected from access by external entities by the isolation mechanism used by the compartment.

Isolation can also be achieved through virtulisation. Virtualisation was first developed in 1960's by IBM Corporation to partition large mainframe computer into several logical instances also called Virtual Machine (VM) running on a single physical

machine. The Virtualisation has two primary benefits: First, shared resources such as memory are virtualised and share the physical resources of the machine. The aforementioned is different to non-virtualised systems where all resources are dedicated to the running process. The second and main benefit of virtualisaion is isolation. Programs running on top of one VM cannot see programs running in another VM. The software managing the physical resources between VMs and responsible for isolation between them is called Virtual Machine Monitor (VMM). The security and trustworthiness of the VMM are essential for enforcing isolation and provide security for VMs running on top of it.

2.3.1 Hardware Virtualisation

The virtualisation architecture necessitates protection from adversaries and attacks (e.g. attacks from a resident VM) for guaranteed trusted execution and adherence to its expected function. When executing under the same privileges, the threat of a VM attacking the VMM is very plausible. An attacking VM can change the code of the VMM and gain unauthorised access to other VMs including memory, code, and data on transit. To counter that, the VMM makes use of hardware extensions such as CPU virtualisation extension and privilege levels numbered from 0 to 3 [84], to run on higher privileged environment than the resident VMs.

2.3.2 Protection Rings

As stated in the previous section, the IA-32 architecture protection mechanism has four privileged levels with the greater number (3) being least privileged. It indicates that ring 0 has the highest privileges. The privileged levels, also referred to as the protection rings, are CPU extensions. In the Intel processor manual it is stated that *"rings protect application or user programs from each other"* [84]. The protection rings provide domain separation mechanisms, thus, allowing user-supervisor separation enforced by hardware. Software running in protection ring 3 mode are less privileged than software running in protection ring 0 mode and has no direct access to platform resources. It can be inferred that for software running in ring 3 mode to access the hardware, it needs to request access from privileged software such as the one running in ring 0 mode. To illustrate this further, Code modules in lower privilege levels can only access modules operating at higher privilege level through a tightly controlled and protected interface referred to as "gate". An attempt to access code or resources in higher privilege level without having the sufficient authorisation causes protection exception. Most computer systems run the OS in ring 0 mode and the application in ring 3. In such module, where the OS in protection ring 0 and applications in protection ring 3, a *Call Gate Descriptor* is defined to allow a call from the unprivileged level (protection ring 3) to the privileged (protection ring 0). The Call Gate Descriptor has access right information, segment selector for the code segment of the called procedure, and an offset into the code segment. On invoking a call, the processor switches to execute the called procedure in privileged mode as

stated in the relevant Call Gate Descriptor and switches back to unprivileged level when it finishes executing the call. Unfortunately, the aforementioned does not mean trusted execution for the call. If the executing call has some software vulnerability, an adversary might exploit such vulnerability to gain access to resources accessed by higher protection levels.

2.4 Trusted Execution Environment

The several privilege protection levels introduced in previous section do not guarantee trusted execution. The aforementioned mechanism is limited and has uni-directional protection. For instance, a sensitive application running in protection level 3 is not protected from a malicious privileged OS running in protection level 0. In addition, for the sake of simplicity and performance, most OS vendors choose to have the OS including kernel, all drivers, and software components running under the same protection level (protection ring 0), making the TCB in the same protection level enormous, with high risk of inclusion of software vulnerabilities. To illustrate the previous point, there are serious implications to large TCB, the most obvious and critical one is the huge attack surface that grows with the size of the TCB. It can also be argued that most software are not security-sensitive and are included to support the functionalities of the software architecture. Consequently, many security-sensitive applications that run either locally or remotely (e.g. applications that use users personal data) may find it hard to trust a large TCB. The aforementioned rises from

quantitative measurement of the TCB and its trustworthiness. The importance of size of TCB has been acknowledged by many pieces of research [85, 86, 87, 88]. The following sections will give brief background on systems that enabled TCB reduction.

2.4.1 TrustVisor

In the pursue to reduce the TCB for applications on Commodity Systems, McCune et al. [62] developed TrustVisor, a hypervisor that leverages the features of modern processors extensions to overcome the tradeoff to simultaneously achieve a high level of security and high performance. The special-purpose hypervisor provides code integrity, data integrity, and secrecy for selected portions of an application. To achieve that, it enables isolation of sensitive code called Piece of Application Logic (PAL) which is achieved by a small code base (6K LOC) that makes verification feasible. In order to guarantee a trusted execution, it leverages TC building blocks such as Dynamic Root of Trust for Measurement (DRTM) and TPM. The physical TPM stores the measurement of TrustVisor code on invoking the DRTM code, allowing for verifying the state of the execution. In order to secure PAL, DRTM-like mechanism is implemented in software and measured by TrustVisor when it is registered. The hardware DRTM is root of trust for TrustVisor, then the same principles are applied to the application layer. The reasoning behind using software for measurement is to avoid the performance slow-down of using physical TPM. Thus, a software micro-TPM is instantiated for every PAL after passing execution to the hypervisor and destroyed when unregistered.

2.4.2 ARM TrustZone

ARM TrustZone is a hardware security technology incorporated in many of the ARM processors. The TrustZone specifications mandate that the physical core has to be separated into two different modes, so called *Normal World* and *Secure World*. This design choice enables separating the system in two parts with different privileges of execution. The *Normal World* is intended for running rich functionality (Rich OS), thus, a large TCB. On the other hand, the *Secure World* is designated for security sensitive software (Secure OS). The isolation of the *Secure World* from the *Normal World* has several benefits; the former allows for a smaller TCB that excludes the rich OS and for apps to run in TEE, where an app is separated from the rest of the system. Thus, the execution is protected from software vulnerabilities of software running in the *Normal World*.

ARM based Android devices have the Android OS running in the *Normal World* and only designated sensitive software (e.g. Network provider application) running in the *Secure World* . Each world has its own memory address spaces, CPU, cashes and memory, however, only the *Secure World* has access to both worlds' resources. To elaborate further, applications running in the *Secure World* which are referred to as *Trustlets*, have access to the entire *Normal World* memory, yet partial access in the *Secure World* . The secure OS acts as an enforcer within the *Secure World*, thus restricting Trustlets accesses. TrustZone technology is available on most modern smartphones, but has arguably not been used to its full potential [89].

2.4.3 Intel Software Guard Extensions (SGX)

A more recent example of a hardware-enforced TEE implementation is Intel's Software Guard Extension (SGX) technology [34, 90, 91, 92]. Intel SGX enables the instantiation of a protected container called *enclave*. The enclave is an isolated region of code and data within an application's address space. Data within an enclave can be accessed only by code within the same enclave. The enclave is able to protect its data using Enclave Page Cache (EPC), a secure storage used by the processor to store pages when they are part of an executing enclave. In order to protect the execution, the enclaves run on the processor in privileged mode called *enclave mode*. It is important to mention that SGX enables instantiating more than one enclave, and each enclave does not have access to other enclaves data (e.g. trustlets in ARM TrustZone). Thus, it allows a form of user-level TEE that is protected from external software, including an untrusted OS.

- **Enclave** - Intel SGX provides hardware features that create a form of user-level TEE called an *enclave*. The enclave is an isolated region of code and data within an application's address space. Data within an enclave can be accessed only with code within the same enclave. The enclave is able to protect its data using EPC; a secure storage used by the processor to store pages when they are part of an executing enclave. The EPC is built from chunks of 4KB pages; aligned on a 4KB boundary and each page has security attributes in the Enclave Page Cache Map (EPCM), an internal micro-architecture structure that is not

accessible by software. It tracks the content of each EPC page, and enforces access control for accessing the pages.

- **Measurement** - a cryptographic hash of the code and data residing in an enclave at the time of initialisation. The measurement is used to verify that the loaded enclave is what the enclave claims it is.

- **Quoting Enclave** -The CPU signs a description of an enclave on behalf of Intel [93]. As for every Certification Authority (CA), if it's compromised, such a mechanism becomes useless. The Quoting Enclave creates the key used for signing platform attestations which is then certified by a backend infrastructure. The key represents not only the platform but the trustworthiness of the underlying hardware. Only the Quoting Enclave has access to the key when the enclave system is operational, and the key is bound to the version of the processors firmware. Therefore, a Quote can be seen to be issued by the processor itself.

2.4.4 Out-of-band Assisted Hardware

A different approach from using CPU extensions to provide TEE is out-of-band assisted hardware such as co-processor or PCIe card (e.g. Graphic Card, Network Card) connected through a PCIe bus [94, 95, 96, 97]. The out-of-band assisted hardware is a separate hardware that resides in the same platform, however, it is not part of the processing unit, thus, outside of the OS TCB. Typically, the aforementioned

hardware has access to the memory and does not interfere with the main execution. In matter of fact, in a Memory Management Unit (MMU) based system the CPU has no knowledge when the assisted hardware accesses the memory. The model of having out-of-band hardware provides an isolated execution from the system and the OS that run on the CPU. As a result, it can be used for many functions. The most obvious one are memory introspection, and trusted execution environment. In addition, the out-of-band hardware can completely offload execution from the main CPU that can execute different software in parallel, thus, it is very efficient for specific applications such as storage and firewall that heavily rely on OS services.

2.5 Software Vulnerabilities

Software vulnerabilities in code continue to appear in software, thus, exploits continue to cause significant damage to users and risk exposure and misuse of their sensitive data [98, 99]. In particular, the consequences of vulnerabilities in server software grow more critical as more users, governments, and enterprises are moving to deploying their data in the cloud [100]. One of the main disadvantages of cloud environment is that users have little choice in choosing the software and the security measures running on the platform in use. In this section, techniques and methods for mitigating software vulnerabilities are discussed.

2.5.1 Software Protection vs Development Practice

The approaches to eliminate exploits of software can be split into two types, development detection practises [101, 102] and software protection [103, 104]. The development practices focus on the security development life cycle to detect and eliminate error in software. Automatic tools used for security testing and vulnerability scanning are also among the development practises. However, running these tools albeit essential does not guarantee security by design and is required each time a new software is produced, including updates, and maintenance. In addition, security testing and vulnerability scanning lack the decision component and require the developer to understand the reasoning behind the outcome of the testing tools. On the other hand, software protection techniques generally depend on programming languages such as PHP [105], Java [106], and .Net [107]. However, they are not applicable to C/C++, which is the preferred choice from a financial and time-related perspectives and can be applied to any new application.

2.5.2 Reduced-Privileged Containers

The principle of least privileged containers [72] calls for dividing the code into compartments with the least privileges necessary to complete their intended function. The aforementioned approach limits the damage that arises from injecting code in one compartment and prevents exploits from leaking information.

As stated before, isolation provides a good mean to protect a container from the

rest of the system and to protect the system from an untrusted container. Consequently, isolation can reduced the privileges of software components in a system. For instance, an OS, that would typically run in higher privileged mode than an application, can tamper with the application data in memory and with the code execution. However, isolating a container from the rest of the system will prevent an OS from accessing memory application space. For example, when application data is encrypted in memory, the OS can not deduce the encrypted data within feasible time.

2.6 Background Summary

Nowadays, operating systems offer isolation primitives that are cumbersome and ineffective to large size application. Such primitives hinder the security of applications due to the large TCB. It also makes the trust and trustworthiness decision of the large TCB ineffective since the spectrum of execution of the TCB is too large and complex for security measures such as formal assurance, manual review, and semantic attestation. Thus, leaving users exposed to exploits of software vulnerabilities. Isolation based technologies explore how to isolate and protect the confidentiality and integrity of the data. In particular, this thesis explores how software partitioning can reduce the risk of software exploits and harden the security of applications.

This chapter provided the background for the rest of this thesis. The following chapters will expand on the relevant aspects of some of the approaches and topics mentioned in this chapter in more depth. The intention is to provide sufficient details

that may contribute to the understanding of the methodologies, frameworks, and approaches in the relevant chapter. This thesis is built upon trusted computing concepts and security principles [72] that motivate the design of robust and secure systems using hardware.

You are absolutely deluded, if not stupid, if you think that a worldwide collection of software engineers who can't write operating systems or applications without security holes, can then turn around and suddenly write virtualization layers without security holes

Theo De Raadt on the statement "Virtualization seems to have a lot of security benefits,"

3

Gaps Analysis of Applications Partitioning

Contents

This chapter defines the motivation for Trusted Code Base (TCB) partitioning and more importantly emphasises *what* data and functionality should developers secure in applications. The aims of this chapter are: First, to define the problem of privilege separation and application partitioning to achieve better security in applications. Second, to shed light on the gaps in research on privilege separation and application partitioning. Finally, to define sensitive functionalities and use-cases that must be isolated and the rational behind it.

3.1 The Problem

Application partitioning has been used to assist privilege separation in application for many years [108, 70, 37]. The ultimate goal is to enhance the security of applications and preserve the confidentiality and integrity of users' data. It is imperative that a large application requires privilege separation [72]. However, there are very diverse approaches to partitioning, and different metrics of considerations to mitigate vulnerabilities. This raises the following questions: what motivate partitioning schemes? What aspects to consider when partitioning? How can vulnerabilities affect the partitioning schemes?

Despite the research done in this area, the problem of application partitioning is not clear. Approaches of application partitioning used in privilege separation are mainly ad-hoc solutions and developed specifically to target a particular type of adversary. In previous research, we have seen examples of a *default-deny* scheme that

is used to repel an attempt of access to privileged operations by adversaries exploiting un-trusted partitions [37, 38], typically adjusted to an application design. In the same way, information flowing from a remote user to a server is secured through its entire execution [109], that may or may not include several partitions isolated from each other. The partitioning would typically consider the trustworthiness of software blocks, performance, limitation of the technology providing the containers, and the adversary that is able to manipulate the interface to the container.

In brief, partitioning is applied differently in securing applications and current approaches do not give strong sense of applicability to various types of applications. For that specific reason, this chapter sheds light on the disperse approaches taken in previous research on application partitioning. Through analysis of previous work, we attempt to classify the different approaches, define trade-offs between factors determining partitions, and analyse the methods used for partitioning. However, this chapter does not discuss the efforts required to partition applications.

3.2 Related Work

Partitioning applications has been investigated and used in many pieces of research [37, 69, 70, 110, 38, 111, 112]. However, despite the wide adoption to such approach, there is no one conclusive study that depicts the rational, methodology, and security guarantees that can be offered by applications when such an approach is applied [113].

In Privtrans [70], static analysis and dynamic flow are used to partition application into *monitor* and *slave*. The key point behind this is to partition the application into two domains, with high privilege given to the *monitor* and low privileges to the *slave*. As previously addressed in other pieces of research [71, 114], the assumption is that the *monitor* is trusted and would perform privileged operations, while the *slave* is not, thus, an attacker can compromise the *slave* and try to exploit the system. An important point to note is that the *monitor* acts as a policy enforcer to all calls by the *slave* such as access to files, system calls, and network. In the OpenSSH use-case, assets and operations related to the private keys and authentication are annotated for the sake of isolation. Once the *monitor* authenticates the user, the privileges of the *slave* are upgraded. The *monitor* identifies privilege arguments and checks the *slave*'s request index against a stored hashtable, if it matches, the *slave* is granted access. Following the aforementioned approach, OpenSSH, chsh, thttpd, ping, openssl, are used as case-studies to protect annotated sensitive data. The protection provided with this approach is preventing unauthorised process from accessing privileged operations before it's authenticated by the monitor, which upon authentication changes the *slave*'s process ID and updates the allowed list to accessing privileged operations.

In Privman [71], Kilpatric et al. present a library for partitioning applications for isolating privileges of trusted code: authentication, accessing a file, and accessing the network. Privman, relies on splitting a process into two higher and lower privileges in which trusted code executes on the higher privilege. However, the library does not address the granularity of partitioning or give use-cases of applications and their

sensitive partitioning. In addition, there is no consideration for the interface to the privileged partitions since the split relies on the privileged partition to enforce policy on the requestors based on a pre-defined configuration. To enforce a decision, the privileged partition authorises access based on the process ID and its status. In this model, a process has to earn access to privileged operations.

In [114], Provos et al. present preventing privilege escalation through splitting a process into *monitor* and *slave*. The privilege part resides in the *monitor* (the parent) and the unprivileged part resides in the *slave* (the child). A *slave* must ask the *monitor* to perform any operation that requires privilege. In turn, the *monitor* validates the request and communicates the result to the *slave* if the request is permitted. Again, the authors clearly state that it is necessary to identify the operations that require escalation of privileges, yet, they do not explain what should be isolated and given higher privileges. It is important to note that the methodology used to implement the communication between the *monitor* and the *slave* is relying on Finate-State-Machine (FSM). The *monitor* is modelled by a FSM that monitors the progress of the unprivileged *slave*. The methodology operates in two modes; pre-authentication phase and post-authentication phase. The pre-authenticated phase is when the user contacts the system through the *slave* and before validation. In the post-authentication phase the user has successfully authenticated to the system and the *slave* has the privileges of the user such as file system access. It can be seen that the methodology of *monitor*/*slave* implements policy on accessing the cryptographic resources and files. The authors present as an example for this approach the SSH

protocol, and they isolate and give higher privileges to the execution of the following blocks: Key exchange, authenticated key exchange, user validation, password authentication, and public key authentication. The *monitor* allows access to these blocks by referring to a pre-defined configuration, and correctness of the input given to it. As a result, the unprivileged *slave* (child) does not have access to sensitive operations and data such as key exchange, user validation, passwords, and keys. In addition, to guarantee permitted use of the cryptographic operations such as key exchange or signing, the user must be validated first and only then its status is changed to permitted user and the *slave* allowed access to the cryptographic operations.

In Wedge, Bittau et al. [37] present splitting applications into reduced-privilege compartments. The partition is based on threads that provide each partition with isolated, protected and policy-based memory access by other partitions. To execute on privileged space, a callgate is called to perform the operation. In the first case study, they discuss partitioning a mail server into three blocks with different privileges: client handler that processes user's input and commands, login block for authentication, and email retriever block to obtain the emails. The split of partitions reduces the privilege of the client handler block and increases the privilege of the login and email retrieval blocks assuming the former two are not exploited. From privilege separation point of view, a monolithic mail server with an exploit can allow access to all process memory including *confidential* data of passwords and emails. It can be inferred that the partitioning choice assumes untrusted client handler and accommodate to this mistrust in the more privileged blocks, the login and email retrieval.

In the Apache/OpenSSL case study [115], the objective of the partitions is to preserve the confidentiality and the integrity of SSL connections (e.g., keys, exchanged messages) and prevent man-in-the-middle attacks. For instance, an attacker with a knowledge of the private key of the server can eavesdrop on the connection. This is a fair assumption as most systems (Linux, Windows, etc) would have the private key unecrypted in memory, thus, an unauthorised kernel module has access to that key. In addition, an attacker with the ability to influence the code generating the session keys can force the server to generate the same key or a key of its choosing. To meet the objectives, Apache/OpenSSL was partitioned into several blocks. The setup session key block runs under higher privilege and it's the only partition that can access the private key and negotiate the session key. The session key is passed to another partition so it can be used by the connection. It can be inferred that less privileged partition that uses the key cannot negotiate a new key and can only use the key passed by the setup session key block. In a second spin off of the same use-case [37], the authors consider a different attacker, an attacker with the ability to act as a man-in-the-middle on the Secure Sockets Layer (SSL) handshake. In this case, preventing an attacker that injects malicious code in the SSL handshake partition from obtaining information and maliciously use the key. The generation of the key is done in privileged state and never leaked to the the SSL handshake block. The SSL handshake partition can complete the handshake but not learn about the session key. On completion, the SSL handshake partition is terminated and the execution is passed to the client handler partition to use the session key for the connection. On the other hand, when considering a different trust model of the client handler being vulnerable

to exploits (e.g., code injection), partitioning should be applied differently. Limiting access to the sessions keys and private key in such case is not enough as the client handler performs the SSL read and write. To solve the former, the SSL read/write are isolated from the client handler and given access to the session key. Thus, the client handler, can only leak a cipher text but does not access the session key or the data directly.

In an additional use-case where the application is designed with privilege separation in mind; the OpenSSH, access to the private key was limited to minimal code. The code handling client's requests prior to authentication runs on minimal privilege and steps-up privileges for the user only on successful authentication. In Wedge, the three security sensitive calls: (1) user password authentication, (2) DSA key based, and (3) S/Key challenge-response were isolated from one another and were allowed access to the private and public keys. The consideration of partitioning the application in such a way is to separate the privileges of those three sensitive parts and isolate between their execution and enforce access to the application keys and configuration. In addition, in order to prevent leak of information such as the known vulnerability of leaking the presence of a user's name [116], the return value from the isolated password partition returned a dummy. Hence, the password callgate cannot leak information if a user's name is present in the system, but rather would return a wrong password. A point often overlooked is that the interface between trusted and untrusted partitions are known to be target of attacks [109, 2] and must be considered when partition applications.

In Crypton [117], a protection for users' sensitive data is designed to protect against client-side web stack. The rational for partitioning the webmail software into two parts is to keep the data protected from vulnerabilities in the browser components [118, 119], browser add-ons, and application components. As previous studies have shown [118, 119, 120], many vulnerabilities have been found in these areas that lead to leakage of sensitive data. Thus, in Crypton, data is stored encrypted in memory and accessed only by the Crypton-kernel process.

In [69], the authors focus on isolating policies through using parent/child design. An HTML5 application is partitioned to one privileged parent component and arbitrary number of unprivileged children. The parent includes the policy related to isolating the children from one another based on an input from the user. In addition, Bootstrap Code and Shim are also included in the parent partition as an entry point to the HTML5 application and to allow privileged calls from the children. On the other hand, the children run the application code and request privileged operation using the Shim interface to the parent. As a result, any library or code with vulnerabilities like Cross-Site Scripting (XSS) can be exploited in a child compartment without affecting the other compartments.

In Slaus [121], Strackx et al. present a secure process compartment based on kernel memory access model. In their work they present partitioning of application to prevent a privilege escalation in the application and confine the vulnerability to a specific piece of functionality rather than the entire application. As a poof of concept, they apply their method to three applications. **X.509 certificate signing applica-**

tion consists of four partitions: a parser, a validator, a signer, and a logger. These components can call each other through an interface that authenticates the caller and the called, such that a vulnerability in the parser cannot cause a complete compromise of the system by accessing all its functionalities, such as access to the signing block. In addition, to protect the private data, such as keys, each compartment consists of public and private parts. The public part includes the interface for accessing the private part and memory cannot be accessed in any other way. Furthermore, to limit the impact of exploiting vulnerabilities in a new compartment, the compartments inherit system call privileges from their parent. **PolarSSL** was partitioned in a way that each connection received a new compartment, thus, limiting any code injection to one compartment and to one user. The rest of the code resided outside the compartment and was referred to as untrusted by the authors. The work by Strackx shows the important consideration of isolating the different data flow of users and recognises the big challenge of defining interface to these compartments which is addressed in section 3.5.2. **Gzip** was partitioned in a way that the parser was isolated from the rest of the application and restricting system calls. Despite inflecting a high overhead for small files, security was enhanced for the application by limiting the impact of vulnerabilities.

In [112], the authors present an automatic partitioning scheme that performs partitioning based on the protection mechanisms that are most needed in each partition. The former is achieved through identifying points of user authentication and data access without access to source code. This is achieved using a *monitor* at runtime

that identifies authentication points, accessing files, as well as events and switches to different execution with adapted defences such as Dynamic Taint Analysis (DTA) and instruction-set randomisation. One of the strong assumptions of this work is that authentication is binary, either succeeds or fails, causing the execution flow of the application to follow a different path in each case. The partitioning of application runtime is divided into trusted or untrusted for each authentication point or data access, based on successful authentication and successful data access, the runtime execution is given higher privilege of execution and ends only when the relevant section of the authentication of the data ends.

In SOAP [35], a different approach to partitioning application is taken. Instead of isolating the sensitive operations, the authors choose to sandbox components that are known to be the cause of exploits of vulnerabilities. In the FreeBSD's fetch application the parser and the network stack are sandboxed in different compartment.

In [38], Marchenko et al. discuss the partitioning of protocol implementations to protect sensitive data. In their work, they confer several principles for partitioning and investigate two use-cases: OpenSSL and OpenSSH. The item of interest for protection is the private key in RSA (when RSA key exchange protocol is used) and the session keys. Two attacks are discussed, the session key disclosure attack and the oracle attack (private key). A root access has any access to these keys, however, it is not the only way to use these keys. In monolithic applications, a compromised part of the application can lead to jeopardises the entire application which can result in unauthorised access to keys or to privilege operations. For instance, when an

adversary places themselves between the client and the server as man-in-the-middle they can intercept all messages between the two ends when having access to the signing operation (using the private keys). To prevent escalation of privileges, the authors propose a three stage partitioning to limit an attacker's capability to tamper with the execution of sensitive operations (e.g., handshake) and place themselves as man-in-the-middle in a vulnerable partition. The three stage partitioning (1) session key negotiation stage, (2) pre-authenticated stage, and (3) post-authenticated stage are applied for OpenSSH and OpenSSL to prevent unauthorised access to the key and man-in-the-middle attacks. To illustrate this further, the partitioning scheme takes two main points into account: the interface to the sensitive data (private and session key) and allowed access to the operations. Thus, the use of the private key is limited to one partition that is assumed to be trusted and the interface to the partition doesn't allow for adversaries to place themselves in the middle using fine-grained operations (e.g., signing key).

In [110], Shreds uses threads to isolate the execution of sensitive operations in five applications. In Curl and Minizip, the partitioning of each of the applications into untrusted and trusted parts is done by isolating authentication code to protect the client's password. In OpenSSL and Lighttpd the credential validation operation is isolated to protect the credentials. In OpenSSL, it isolated access to operations accessing crypto key. The important thing to mention about thread is the Lines of Code (LOC) of the isolated regions. The LOC size of the isolated regions of the use-cases are between 7 and 526 which is little compared to approaches using processes

or enclaves. The disadvantage of such an approach is the simplicity of the interface to the isolated regions, which allows an adversary to attack and impersonate. As mentioned previously, there is no need to obtain knowledge of a key, it is enough to have access to privilege operations to perform an attack.

In [111] the authors present SecureKeeper, a secure Apache ZooKeeper [122] that keeps the confidentiality of users' provided data which is stored in the ZooKeeper database. The key used to encrypt/decrypt the data in the database is revealed only to the client after successful authentication between the client and the server. To guarantee that privileged software has no access to the key prior to connection establishment, an enclave is allocated to each client on the server where it handles the communication with the client (also called entry enclave). It also encrypts/decrypts the traffic between the client and the server using unique session key per client that are solely available to the enclave. Upon a successful establishment of a connection, the client dedicated entry enclave gets access to the database's encryption/decryption key which is used by the enclave to encrypt/decrypt data from the database. To protect the data stored by clients, two inputs are defined as sensitive: the payload and the path information of a znode. Due to an adversary's capability of mixing and switching payloads and znode names, the znode path and payload must be bound together, hence, the reasoning behind defining the znode path as sensitive.

As can be seen, there are a myriad of approaches and attempts to address the application partitioning challenge. However, most fail to give the clear definition of the considerations that should be made in order to make those approaches widely

applicable. Furthermore, many of the previous work do not necessarily cover all potential risks to the system.

3.3 Sensitive Data/Functions

The question of *what* data and function to isolate in a separate TCB have been the focus of many pieces of research. The rational behind partitions is separation of privileges, untampered execution environment that is protected from the rest of the system, and smaller TCB. However, the sensitivity functions and data in applications varied and addressed different adversaries and types of attacks. The following list summarises the targets of protection.

- **Cryptographic Operations:** Cryptographic operations are the backbone for the security of the system. For instance, a change in the implementation of a hash function even by merely flipping one bit in the hash function (e.g., SHA1, SHA256) can lead to unreliable results that are susceptible to brute force attack in reasonable time, making the attack on a system feasible. For example, setting up a session key, and signing operation [37].

- **Private Data:** such as emails and files, mainly privacy and secrecy.

- **Cryptographic keys:** secret assets that are used to protect security and sensitive content such as session key and private key in OpenSSL, and encryption/decryption key for databases such Apache Zookeeper.

- **Policy Enforcer:** access control to assets and functions using platform state, password, location, and predefined access list to platform components [69].
- **Protocols:** protocols used to obtain or exchange secrets such as key exchange protocol, and handshake protocol.
- **Obfuscation:** Obfuscation is the wilful obscuring of the intended meaning of communication by making the code and data difficult to understand, usually with confusing and ambiguous language.
- **Authentication:** verifying user or process identity such as login in order to escalate the privileged to allow performing privileged operations [70].
- **Authorisation:** who is allowed to perform authorised operations such as send and receive [37].

3.4 Issues with Partitioning

It is evident by now that application partitioning reinforced by containers can enhance the security of applications by mitigating software vulnerabilities and rebel strong adversary who has control over the Operating System (OS). However, there are non-obvious trade-offs to consider. A monolithic application secured by one container can prevent an adversary who has control over the OS from interfering with the execution. The adversary can tamper with the interface to the application or data received by the application. However, by carefully designing the application (e.g.,

encrypting the input before passing it to the application and decrypting it within the application), the developer can mitigate such attacks. Nonetheless, such an approach does not consider software vulnerabilities within the applications themselves. When the number of LOC in an application is within tens and hundreds of thousand, it is hard to assure trustworthiness. In addition, one needs to take into account the limitations of the technology, for instance the technology used to provide the container may have limit on the size of the container, in such case it's inevitable but to accommodate such a limitation by moving away from the monolithic structure. The other thing to examine is how the use of the container technology affects the execution of the application in relation to latency, performance, and throughput. When attempting to reduce the impact of software vulnerabilities in applications, an application may be partitioned to multiple sensible partitions to reduce the TCB in each container. In that case, a software vulnerability such as a buffer over-read in one container or outside the container will not affect other containers. However, the OS has a good view of the execution inside an application, and it can observe the interface to/from the container, thus, an adversary can still manipulate the interface to attack the application. Needless to say, using containers adds to the time of execution, thus, a partitioning scheme must take into account a normal application performance.

It can be inferred that when performing partitioning there is a battle between three factors: (1) normal application performance, (2) mitigation of software vulnerabilities by splitting applications, and (3) protection from an adversary who has control over the OS. Figure 3.1 represents the three attributes of a partitioning schemes.

It might be straightforward to satisfy one or two of those characteristics, and it's most likely that most approaches using containers technology will satisfy one of these requirements or the two characteristics of *Normal Application Performance* and *Protection from the OS*. However, achieving a complete protection that includes all three components has not been introduced thus far.

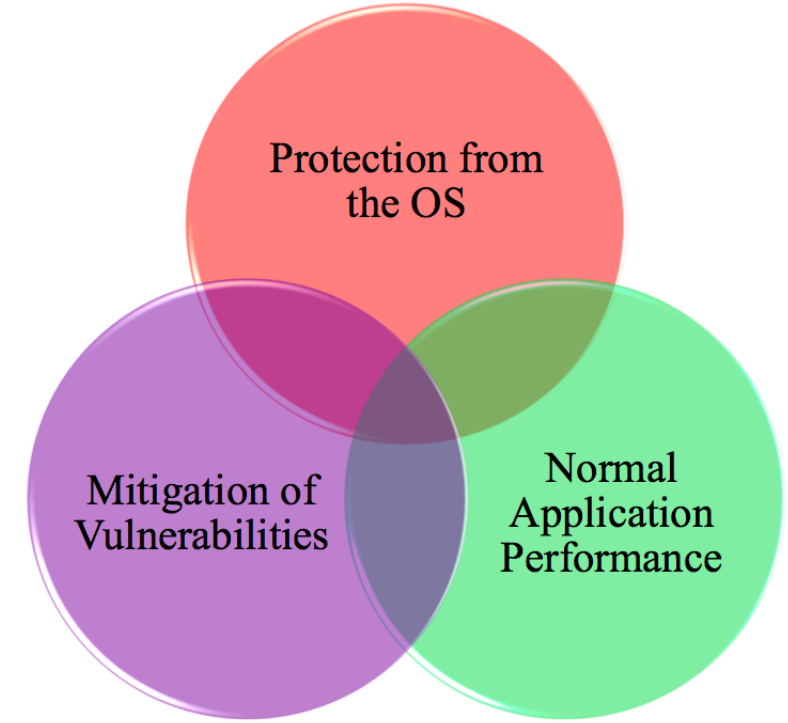

Figure 3.1: Venn Diagram of Partitioning Schemes: The Venn Diagram describes the three relevant characteristics to look at when isolating software based on a partition scheme. Isolating an entire application inside a TEE, the "protection of confidentiality and integrity of an application from the OS" characteristic is obtained by applying such scheme. Isolating an application into two parts each inside a separate TEE the "mitigation of software vulnerability" is obtained. The overlap between the circles represents a partitioning scheme that takes into account two or more characteristics. The size of the overlap indicates the likelihood of a solution to cover the relevant characteristics. Partitioning an application into two partitions, each inside separate TEE and considering the communication channel, interface (to and from the TEE), and attestation, the two characteristic of "protection from the OS", and "mitigation of software vulnerability" are obtained.

3.5 Gaps Analysis

This section presents the motivation for privilege separation with application partitioning and aims to group and formalise the aspects of privileged separation.

Table 3.1 summarises the previous work done on privilege separation by partitioning application into compartments. It can be inferred that the main purpose of partitioning into privileged and least privileged is to protect the data by restricting the information flow. Three types of information flow have been observed. The first and most common type is the *Master-Slaves*. In this type, the *Master* resides in a privileged compartment and restricts access to reduced-privileged compartments (*Slaves*). Furthermore, it enables flexible control of the privileges of the slaves [70]. The second type is the *Default-Deny* initially presented by Wedge [37]. In this type, the application is partitioned into several trusted compartments and one untrusted compartment. The rational behind this partition is as the name suggests, other compartments are always assumed untrusted, thus, prevented by design from accessing privileged operations. The third type is based on Decentralised Information Flow Control (DIFC) and seems to get more attention in the security world in the last decade. In this type, the rational is isolating between the different flows (e.g., per user, per session) to guarantee untampered execution and separation between the different flows.

3.5.1 Citation Tracking Tree

In order to capture a sufficient view on privilege separation and application partitioning we setup our investigation as follows. The first research work on application partitioning was introduced in 2003 through Privman [71], which used Thttpd and WU-FTPD as case-studies. Privman constitutes the research root for this current study. Since its introduction, much work has been done on privilege separation. Figure 3.2 illustrates the three time eras considered starting from Privman and branching into two additional eras. Privman in 2003 constitutes the root to the second era which ranges from 2003-2008 and are directly citing Privman to lay the grounds for their work. The third era covers 2009-2016 and presents an advancement on top of the work done in the second era. This chapter covers all the time-line from introducing the first milestone in application partitioning in 2003 through works that have built and advanced on it all the way through 2016.

Partitioning applications into privileged and less privileged compartments can be classified into several categories. For the sake of simplicity we will consider two here: One classification considers application partitioning from attacks point of view [37, 38], where the focus is on a block in an application being compromised and how partitioning into privileged and least privileged compartments can limit an attacker's ability to obtain sensitive information or access to privileged operations. The model used is referred to as *Default-Deny* since the privileged block assumes communications with malicious and untrusted blocks. In another classification [71, 69], the focus

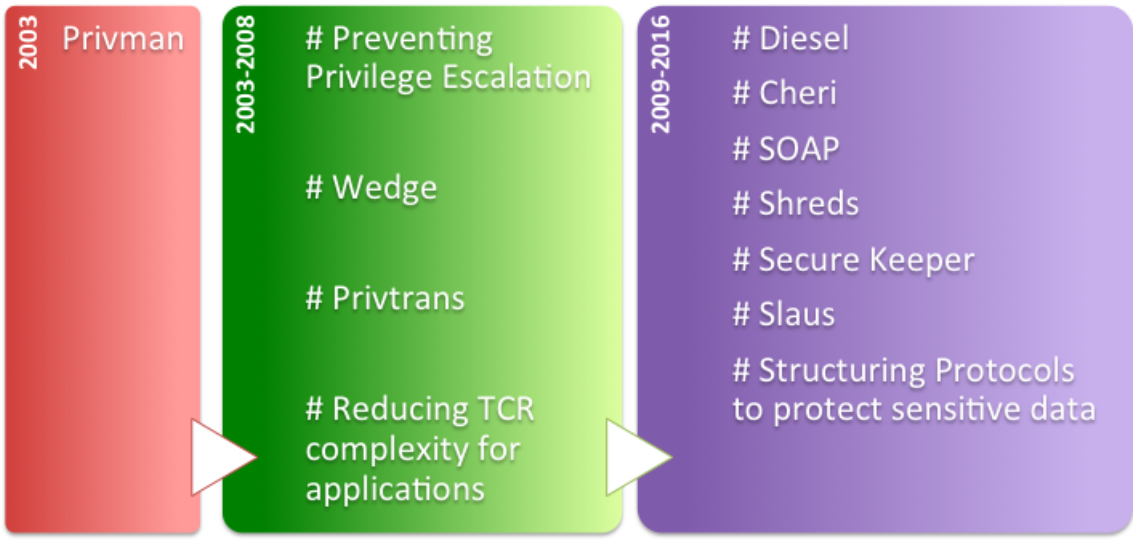

Figure 3.2: Citation Tracking Tree: The Research Tree is inspired by a method known as citation tracking [123, 124], which is a systematic approach to evaluate the impact of a study by tracking its citation record. This scheme shows the pioneering work on Privilege Separation which we followed by tracking the subsequent papers that cited it to define the first branch then the papers that stemmed from citing the later.

is preventing an attacker from accessing privileged operations and enforcing policy using centralised information flow. The aforementioned adopts a *master-slave* type of partitioning by defining a *master* compartment with higher privileges and reducing the privileges of the *slave* compartments.

As for choice of applications used for privilege separation, Figure 3.4 presents applications of interest to the research community starting from 2003 until 2016. The figure shows the number of research studies that used a particular application in their work. It can be seen from the figure that OpenSSH and OpenSSL are the most attractive applications for privilege separation studies. That is largely because the software design of these applications has security in mind. For instance, every new user session is handled in a new process to isolate the execution per user. The

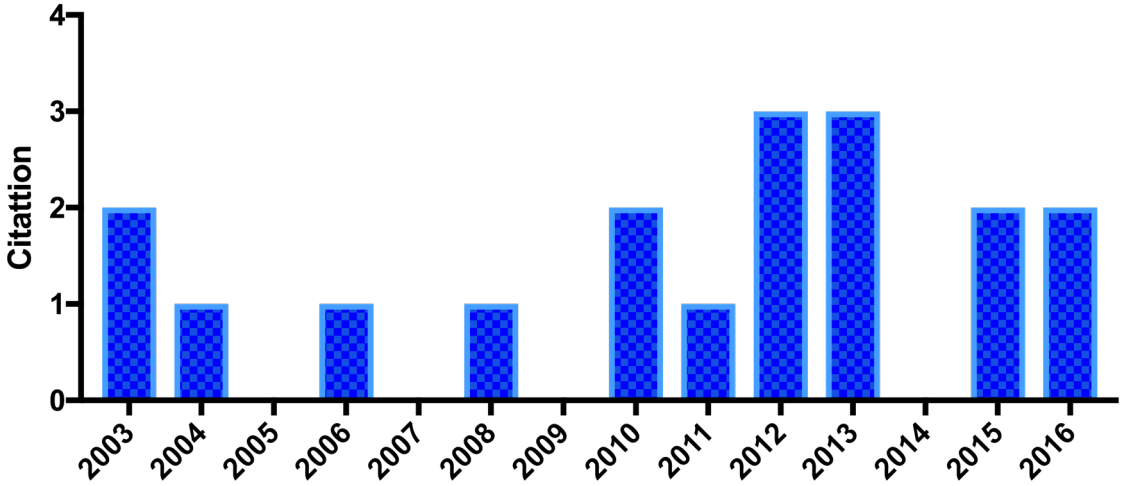

Figure 3.3: Distribution of privilege separation research. This histogram describes the number of privilege separation research papers cited per year. The graphs shows the growing interest in the field. In 2015 Intel SGX was released, thus, the expectation is to see more work on privilege separation in the coming years

spread of applications used include storage, communications protocol, web servers, compression, browsers, browsers plugins, web management software, and text editors. It can be inferred from the figure that four communication protocols are of interest in research. Another major interest is web servers and mail clients, while databases application like Apache Zookeeper, SQL, and Memcached are only getting more attention in the last decade.

3.5.2 Interface to Container

The interface to isolated regions is an essential element in privilege separation. A partition might be a secure asset within a container, however, it is not trivial how to prevent unauthorised access to functionalties using that asset, nor how to obstruct the external calls from leaking information about the flow of operations. For instance,

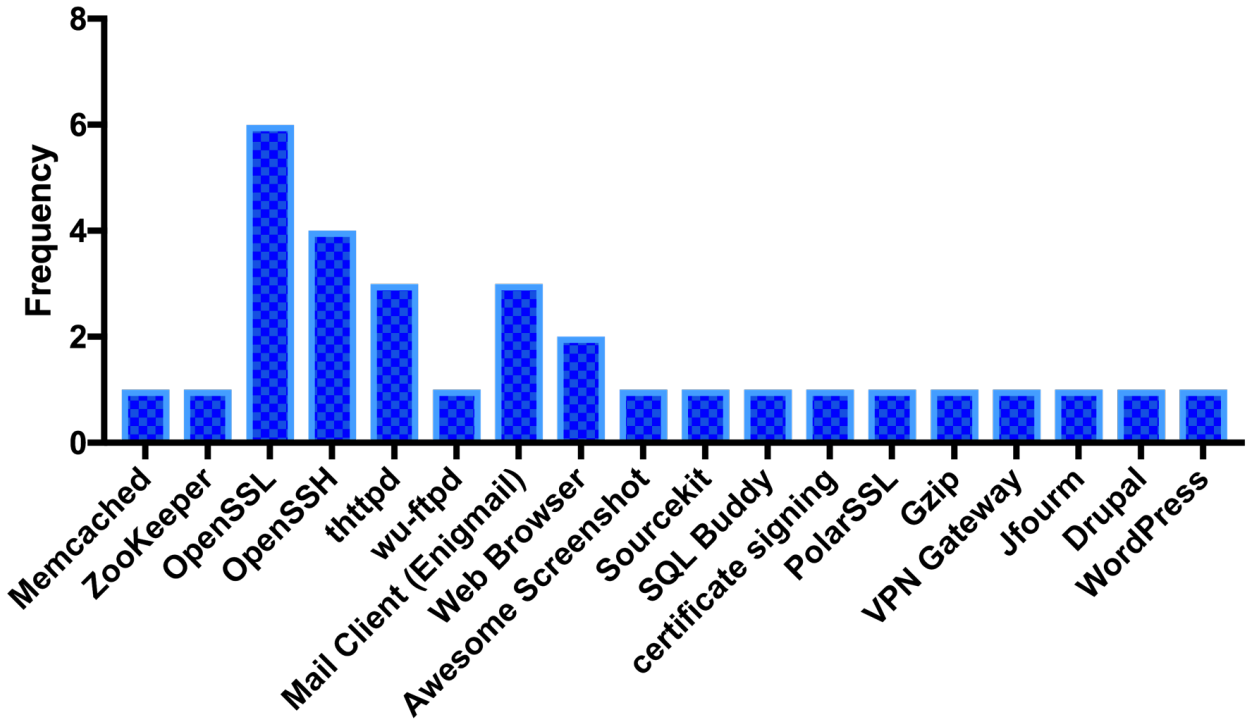

Figure 3.4: Distribution of Applications. The Figure shows which applications are of most interest in the field of privilege separation. OpenSSL and OpenSSH are the most used as case studies with other applications scoring lower. It is important to realise that when considering vulnerabilities, OpenSSL, OpenSSH, httpd, and Mail Client, the highest cited applications, have the most vulnerabilities reported on CVEDetails [125]. Thus, it justifies the massive adoption in privilege separation work. However, it begs the questions: when vulnerabilities are of such interest, why are there so little experiments on privilege separation and partitioning schemes that discuss reducing the impact of software vulnerabilities?

Marchenko et al. [38] discuss a partition that prevents direct access to the private key in OpenSSL, yet, allows access to signing operation using the key. This section states the three main points to consider when considering the interface to a container.

Robust: Isolating the execution of sensitive software partitions can guarantee unhampered execution. However, an adversary can exploit the interface to the partition to perform various types of attacks. For instance, in SecureKeeper [109] the authors discuss an important design choice that rises from the partitioning. In Zookeeper, a sequential number is added to the path name on creation of a new node. The sequen-

tial number increases on node creation and is used only during node creation. In order to prevent a misuse of the sequential counter, only authorised clients authenticated in a separate compartment can perform creation requests.

Authorised Access: Privileged operations should be accessible to authorised requestors only. When privileged operations are granted to an adversary they can use it to perform an attack on the system. For instance, when an adversary is allowed access to a sign operation, despite not having the key, they can use privileged operations using the key to hijack a connection. Specifically, in the key exchange protocol used by OpenSSL, the server signs data and sends it to the user for validation. An adversary can place themselves in the middle between the user and the server and send a signed data [38, 37]. It can be inferred from previous work that there are two ways to handle unauthorised requests. The first approach is adopted by Wedge [37] by defining a secure interface between trusted and untrusted partitions to prevent reduced privilege partition from requesting sensitive operations (called security by design). The second approach, which is mainly used by all approaches using *master-slave* type of partition, is to enforce policy based on the requestor and its state (pre-authentication/post-authentication) [71, 114, 69]. For instance, when a *slave* process requests privileged access from the master, the master checks its status (if it's authenticated to the master) and whether the request is allowed.

Least Privilege: A partition should not be given high privileges but least privileges to perform its operations as it has been advocated by Shroder and Saltzer [72], but also noted in various other works. This is indeed the case in many applications where

prior to escalation of privileges of a partition it needs to earn the right to be higher privileged. The aforementioned is different from the principles of privilege separation [72] since the privileges are not predefined but rather earned. For instance, in [38] they proposed using DIFC and one partition per client to establish the session key, in what is known as *noninterference* [17]. The partition per client is not given privileged access to operations before authenticating to the application, only then it earns the new privileges. A similar approach is considered by others where an application is partitioned into *slaves* and *master* and the *slaves* are granted access upon successful authentication only [70, 114, 37].

3.5.3 Information Flows

In the related work subsection of this chapter, I discussed several papers dealing with how to secure sensitive operations and perform privilege separation between different partitions in the same application. Traditionally, the OS separated privileges between trusted and untrusted processes. However, it can be seen from many attempts to partition applications [37, 70, 109] that even within applications there is a need for privilege separation and protection for information flow. In reality, moving away from monolithic applications and assuming strong adversary such as the OS, introduces information flow issues between trusted and untrusted partitions. Although isolation technology like Intel Software Guard Extension (SGX) can protect the execution of a partition inside an enclave, if the interface to a partition allows access to privileged operations (e.g., signing operation), then the untrusted partition can use this infor-

mation to impersonate the trusted partition. In addition, as have been discussed in [109], in some cases, to protect the information flowing out of a trusted partition, several trusted partitions need to be placed to support secure information flow. To achieve that, applications need to adopt decentralised and centralised information flow when needed to prevent leaks and mix of confidential data between the different users.

When considering partitioning, it is essential to distinguish between the different flows of execution that should not interfere with each other or have information flow from one user to another, which is particularly relevant in multi-tenant environemnts. Applications concerned with multiple users should isolate between the different execution flows (e.g., users) and protect the information flow of each user/session. Thus, adopting DIFC [126], specially when the protection between the flows is enforced using TEE, can report on the trusted and protected execution of a certain flow. For instance, a storage application that handles many users in parallel can isolate between the different users by defining an information flow per user/sessions that is isolated strictly during execution and in memory.

3.5.4 Secure Isolated Region

Isolating a software partition inside a container guarantees smaller TCB as the software partition inside the isolated region is protected from external software including privileged one. However, that does not give any guarantees or trustworthiness of

the software running inside the container. The smaller TCB, however, does make it feasible to verify the trustworthiness of the software using formal methods. Unfortunately, up to date none of the approaches used tools to give assurances of the viability and trustworthiness of a giving partition. In [127, 128], the authors present a design and verification methodology to formally assert that a secure isolation region (e.g., enclave) does not leak information and keeps the confidentiality of the secrets within the enclave. To achieve that they use formal methods and define principles to be used when verifying the secure isolated regions. The methodology is tested on the largest MapReduce examples: Revenue, IoVolumes, and UserUsage. When partitioning the TCB of an application, it is critical to verify the property of the partition and make sure it does not contain vulnerabilities. An adversary able to exploit a vulnerability inside the isolated region makes the partitioning efforts and the protection from external software ineffective.

3.5.5 Containers

As mentioned earlier in this section, containers are used as a mean to separate privileges whether it's privileged or unprivileged containers. There are three main abstractions used as containers. The most well-known is separation using processes, in which each process acts as a container and the separation and connection between the processes is achieved using the kernel. The second abstraction is threads, an abstraction of a processes. The isolation of a thread is typically achieved with the help of the kernel and applying compiled run-time restriction [110]. The third ab-

straction is achieved in hardware and is called *enclave* by Intel SGX and *secure world* by ARM TrustZone. The latter allow execution in TEE that is isolated from the rest of the system. Unlike containers that are based on processes and threads, the TEE protection is based on hardware.

3.6 Gaps in Research

3.6.1 Partitioning Schemes

Previous work on splitting applications into least privileged and privileged compartments was motivated by either enforcing policy on accessing privileged operations based on status and identity, or based on assumptions of which software blocks might be attacked, or securing an information flow from the user through all stages of execution and back to the user. Each of the aforementioned schemes aimed to rebel specific types of attackers. In addition, each saw fit to place different number of compartments and different partitions in order to secure privileged operations and sensitive data. This begs the question: what are the necessary compartments? What blocks to isolate? Which blocks have high risk of vulnerabilities? Assuming that denial-of-service is not the aim of the adversary. It is hard to define an optimal partitioning scheme that would meet all the issues discussed in section 3.4. However, it can be argued that different partitioning schemes can satisfy different requirements. Future research should focus on defining different partitioning schemes against requirements.

In addition, the diversity in applications makes it impractical to define general partitioning schemes that are suitable to all types of applications. It can be seen from previous work that communication protocols are the most used in privilege separation research. However, it is only in recent years that storage and map reduce applications have become of interest. Examining a sufficient number of applications can yield different schemes that are not only dependant on a general sensitive operation as defined earlier, but are also based on the type of application and the different software blocks it has. Future research should focus on defining the differences between applications and the different requirements from each scheme.

Nevertheless, reducing the impact of vulnerabilities is the most undervalued area concerning partitioning. It can be argued that compartmentalisation can reduce the impact of exploits of vulnerabilities in applications. This has been discussed in several manuscripts [37, 38, 1, 4, 35]. Many of the vulnerabilities in applications are memory related. Hence, when using appropriate memory isolation technology (e.g., Intel SGX) it is possible to draw a clear line between compartments. Even with the presence of a strong adversary that is capable of reading every bit in memory, upon using Intel SGX, the compartment memory data is encrypted when leaving the processor and decrypted on entering, making it impossible for the adversary to access it. Our paper discusses the potential of reducing the impact of several types of vulnerabilities [1]. However, further research is needed in this area to investigate how to take advantage of hardware compartments to mitigate software vulnerabilities. In addition, while there is no absolute possible way to point out to every vulnerability in an application, it is

possible to speculate where vulnerabilities might reside in code. That knowledge can guide the research on partitioning schemes.

3.6.2 Formal Verification

Privilege separation with application partitioning can reduce the impact of software vulnerabilities. However, as it can be inferred from this chapter, a sensible partitioning scheme would consist of isolation of several partitions in multiple containers. In the last few years, researchers have started looking at properties of isolated regions in order to assure the trustworthiness of the software inside a container. The work done by Sinha et al. [127, 128] is an undertaking towards sensible design that utilises containers. However, the aforementioned work is too restrictive for some applications and future work should aim at investigating additional properties of isolated regions while allowing flexibility and guaranteeing trustworthiness. Furthermore, there have been no research on formally verifying partitioning schemes. A partitioning scheme would comprise of several containers instantiated under the same process and would typically have an interface that can be observed and tampered with by the OS. Future research should focus on verifying the interfaces between the different containers and the likelihood of a successful attack. Such research is only possible once a partitioning scheme is clearly defined for an application.

Table 3.1: Summary of previous work: the related work is ordered by year from the oldest to the most recent, and it captures the relevant details about privilege separation. When discussing privilege separation, several question come up: What primitive is being used to enforce isolation (container)? What is the target of isolation (sensitive data/code)? What is the pattern of isolation (information flow)? The table presents all those aspects for each of the papers in the related work

	Year	**Sensitive Data/ Functions**	**Information Flow**	**Container**	**Software Vulnerabilities**	**Use-cases**
Kilpatric	2003	Authentication Policy	Master-Slave	Processes	Escalation of Privileges	thttpd WU-FTPD
Provos	2003	Authentication Crpytographic operation	Master-Slave	Processes	Bypass Authentication Privileges Escalation Information Disclosure	OpenSSH
Brumley	2004	Policy Authentication	Master-Slave CFG	Processes	Information Disclosure Escalation of Priveleges	OpenSSL, OpenSSH thttpd,chsh
Bittau	2008	Authentication Private Data Crpytographic operation	Default-deny	Threads	Information Leakage Unauthorised access	POP3 Server OpenSSL OpenSSH
Marchenko	2010	Authentication Crpytographic operation	DIFC	Processes	Information Disclosure Escalation of Privileges	OpenSSH OpenSSL
Akhawe	2012	Policy	Master-Slaves	Processes	Privileges Escalation	Awesome Screenshot SourceKit SQL Buddy
Geneiatakis	2012	Authentication	DIFC	Processes	Bypassing Authentication Information Disclosure	OpenSSH,MySQL Samba, SVNserve VNC, pure FTPd
Strackx	2013	Authentication Cryptographic operations/keys	Default-deny	Processes	Information Disclosure	PolarSSL Gzip Certificate Signing
chen	2016	Authentication Cryptographic key	Default-deny	Threads	Information Disclosure	OpenSSH,OpenSSL, Minizip,Curl,lightpd
Brenner	2016	Authentication Cryptographic operations/keys Private Data	DIFC	SGX Enclaves	Information Disclosure Escalation of Privileges	Apache Zookeeper

3.7 Automatic Partitioning

Partitioning applications into several compartments is very hard and requires significant knowledge how the application is built and whether it is possible to break it into several blocks. One potential approach to try to ease this process is the use of automatic partitioning, which will be discussed briefly here.

There have been many different approaches to automatic partitioning that differ in the way they define the privileges, and enforce isolation between privileged and unprivileged code. In Privtrans [70], partitioning applications is achieved using Control Flow Graph (CFG) using annotations, which is built by the compiler in the optimisation state. Privtrans partition an application into two parts: a *Monitor* and a *Slave*. The privileged code is put inside the *Monitor* process and the unprivileged code is put inside the *Slave* process and using CFG, the *Monitor* grants/denies access to the requesting *Slave*. Thus, the isolation between the two processes is achieved by the OS.

Using CFG, a state machine of privileged calls is created and handled by a separate process referred to as the monitor. From the CFG, a new graph is built by removing all edges that do not lead to a privileged code. Next, all unprivileged calls are removed which leads to a direct graph of privileged calls. The product call sequences are saved and used as policy rules, to allow and deny access to the *Slave* requests. For example, when a slave process requests a privileged operation, the monitor checks if the call is legit by checking the proceeding call in the FSM.

This approach, is very limited when considering the three characteristics mentioned earlier in the chapter. The first point to make when considering the first characteristic, mitigating of software vulnerabilities, is that there is no consideration for software vulnerabilities in code such as code injection, or attacks such as man-in-the-middle. However, such design does protect sensitive data of privileged operations from buffer over-read/over-flow vulnerabilities in the *Slave* process, and the protection is achieved by the OS. This would all fail though, if the adversary is able to circumvent the OS.

It is hard to comment about aspects relating to the second and third characteristic; protection from the OS and native application performance respectively since their trust model assumes full trust of the OS and no performance measurement are available.

In another work [129], Android Secure world of ARM TrustZone is used to secure sensitive operations in applications such as Google Authenticator. The methods used for the automatic partitioning are code annotation and static taint analysis. This work, is similar to the work done in Chapter 4 where the isolation targeted Trusted Computing functions of the TPM. In the aforementioned work the programmer is required to annotate sensitive data and functions, this would act as a source for the tainted analysis code. For the sink, the authors use SuSi [130], a machine-learning approach that classifies Android sources and sinks extracted from the source code. In particular, the approach is used in Android to generate a list of sinks for OS services such as files, network, and connections. Also, methods that return encrypted data

and methods that are related to app lifecycle such as passing information between apps, and app events. Eventually, when the tool is applied, it isolates small number of LOC on the scale of few hundreds. The isolation of the extracted code is achieved by the *secure world* of ARM TrusZone.

For the most part, this work maintains native app performance while securing sensitive code and data. It can be seen from the performance numbers that there is a minor overhead. That's largely due to the architecture of the TrustZone, which has its own resources such as memory and cores. This is different from Intel SGX that shares the same memory and cores but encrypts data when its in memory.

The previous partitioning scheme is good in identifying sensitive code, however, when sensitive code is 150 lines of code, it can be achieved by intermediate programmer. In addition, using TrustZone provides protection for the sensitive data and code, thus, any vulnerability within the application can not affect the *secure world*. Conversely, the mobile applications used in this work are very pointed toward specific functionality, which is not the case for most applications. An application might have more complex design of more sensitive data and functionality, such as initiating a secure connection, authentication, parsing and others. When present in the same app, each needs its own isolation and protection from other parts.

The last point to make, is the protection from the OS. When isolating sensitive code, the size of the TCB is important and should be as small as possible. However, this is not the case when partitioning monolithic applications, as discussed extensively

earlier and pointed out by others [37, 38]. One of the main things to consider is protection from the OS, thus, smaller code means, less execution in the trusted region and more execution in the untrusted regions. Which can introduce easier point of attack for a strong adversary that has control over the OS.

In Glamdring [131], the same method of code annotation, and static data flow analysis are used to support partitioning. Here the authors try to satisfy two out of the three characteristics proposed in our Venn Diagram 3.1, protecting the confidentiality and integrity of sensitive code and data from the OS, and native application performance. The approach taken in this work is similar to the approach of Rubinov et al. and uses Low Level Virtual Machine (LLMV) and Clang to extract the sensitive code and data by defining the source and sink. Unlike previous work on automatic partitioning, the authors reconstruct the source C files to suit the enclave programming template of split between two environments, trusted and untrusted. The LLMV and Clang allow for a large degree of flexibility of auditing the original source code. Once the code is annotated by a programmer and the source and sink are defined, LLMV and Clang can build the CFG. From the CFG, it's possible to add methods that operate on functions and data, and reconstruct new source files by extracting all annotated privileged functions and their dependencies from the original source files. Clang then allows reconstruction of new C source files of the sensitive data which can be imported in the EDL file (the file that defines the sensitive code and data, and entry and exit functions to an enclave). Of course, the original C files need to be modified to the new enclave functions, but LLMV and Clang provide many plugins

that allow all sort of code manipulation and code modification.

In a last step, once the new structure is defined, the new environment is compiled with the new source files and SGX dependencies as most advertised Intel SGX samples [132]. Glamdring has some other security features that include static backward slicing that allow protection of the integrity of output data. However, it is is out of the scope in this chapter.

In a matter of fact, the results from Glamdring confirm many aspects of the theory on application partitioning discussed here. When considering Table 1 and 2 in the paper and considering the LOC of an application like Memcached, at first, the overhead might seem too large with drop of 40% - 60% in the throughput. However, the additional code of security hardening and Intel SGX Software Development Kit (SDK) constitute 72% of the overall code inside the enclave. For this reason, every instance of an enclave in the proposed approach would have such overhead.

Despite all of the above, Glamdring does not consider mitigation of software vulnerabilities. The current design protects the confidentiality and integrity of sensitive code and data from external software which is provided by Intel SGX. To elaborate further, in LibreSSL case-study it can be seen that the trusted part includes libcrypto and libssl. Thus, in the presence of a vulnerability such as the Heartbleed, there would be no measures to reduce the impact. Not to mention that currently when using one enclave there is no method that can satisfy the *noninterference* property discussed before.

The results in Glamdring confirm that it is possible to satisfy two characteristics with such partitioning scheme, protection from the OS and acceptable native application performance. However, it is important to note that the applications used as case-studies have small identifiable sensitive code, thus, the split of two environments is very logical. In large scale applications, such partitioning scheme may result in very large TCB. For this reason, when considering all aspects there is no other choice but to partition an application into multiple trusted environments.

3.8 Summary

Applications partitioning into multiple compartments have been of interest to the research community for many years. Given the analysis in this chapter it is clear that various approaches have been adopted to perform application partitioning for privilege separation, all of which provide an improvised and very specialised solutions. When partitioning applications into compartments, there are three important aspects to consider: (1) the properties gained using containers, whether it's a process, thread, or TEE (Intel SGX, Arm TrustZone), (2) the trusted and untrusted partitions and the interface between the two and (3) the trustworthiness of the *trusted* partitions. There are three approaches to partitioning applications into privileged and unprivileged compartments. The most common approach is the *master-slaves* which includes two types of partitions of trusted master and untrusted slave(s) in one of multiple partitions. The *master* compartment has the higher privilege and grants access to

the *slave* based on policy and status. In another approach, a *default-deny* approach is used to prevent attacks and leak of information. It is achieved by defining trusted and untrusted partitions and restricting access by untrusted partitions to sensitive data. Thus, an untrusted compartment does not have an interface to a sensitive data or privileged operations by design. The third approach is using centralised and decentralised information flow control by isolating sessions into different compartments. Unfortunately, when comparing between the different solutions, it can be inferred that there is an absence of methodology and schemes for application partitioning into compartment. While it is possible to group the work done into three different categories, it's very hard to extract a pattern for partitioning to protect or even prevent an attack by strong adversary that has full control over the system. Future research should focus on analysing the security aspects in non-monolithic applications as reasoned in previous work by us and others [4, 38, 37].Furthermore, researchers should investigate and develop methodologies for partitioning schemes in relation to attacks. Through understanding of applications, designers can architect sensible partition (as achieved by Wedge [37]). However, even with decent understanding, partitioning is still a highly complex procedure and a methodology can ease the process and save time for developers, one of which can potentially rely on automatic partitioning but further research is needed to fully appreciate what considerations should be made to employ such an approach.

With new isolation technologies and solutions on the rise, it is essential to make use of their full extent rather than adopting an obvious solution of isolating an entire

application inside a container [133]. As mentioned earlier in this chapter, one of the reasons for the failure of Intel protection rings is the inefficient adoption of the four proposed rings. Users and designers of new isolation technologies need to prevent another misuse of what potentially could be a security solution to many of the security issues we face today.

If someone steals your password, you can change it. But if someone steals your thumbprint, you cant get a new thumb. The failure modes are very different

Bruce Schneier

4

Securing Server-Client Authentication Protocol using TPM

Contents

In the previous chapter, the need for sensitive operations such as cryptographic operations to be isolated and protected from the main Trusted Code Base (TCB) to avoid tampering with the execution was discussed. This chapter focuses on securing the execution of authentication protocols in two worlds split environments: trusted and untrusted. First, I present a new client-server trusted authentication protocol based on trusted computing. Second, I present how to secure the proposed protocol by splitting the application into a trusted part running in the *secure world* of ARM and an untrused part- a rich Operating System (OS), running in the *normal world* .

Smartphones are being increasingly used to perform sensitive operations such as mobile banking transactions, storing sensitive data, and as a payment method. For applications to securely perform operations they have to rely mainly on the security provided by the operating system and third-party cryptographic libraries. The aforementioned give no choice to developers but to use the features offered by the OS provider and to trust external security libraries provided by other developers. The latter leaves many developers limited in functionality to design an application that is secure and usable [134, 135]. In order to secure operations, assure the integrity of the application, platform, and the software running on the device, trusted computing approaches have been adopted [135, 134, 136, 137, 138]. In many pieces of research, desktop and server systems were developed to assure the integrity and confidentiality of the code/data within an application [139, 140]. These systems use hardware elements, such as Trusted Platform Module (TPM) and Intels Trusted Execution Technology (TXT) [141]. Most of the proposed solutions are feasible on desktops

and servers but not on mobile platforms and that is due to the absence of Intel TXT or hardware TPM chip in such platforms. Alternatively, ARM based chips are being used in mobile platforms such as Nokia Lumia 830, Samsung Galaxy 6, Iphones, HTC, LG, and others. In which, ARM provides a Trusted Execution Environment (TEE) called ARM TrustZone [67] that gives hardware memory protection for the trusted environment. The latter allows running smaller TCB in an environment that is isolated from the rest of the execution. In smartphones, several pieces of research used emulated TPM. However, these approaches used a software TPM either in the kernel level [142, 143, 144] or as an embedded library [145, 146, 147], but not in a trustworthy environment that is separated from other code in the system. Starting from Windows Phone 8.1, all Windows Phone devices include software TPM. The TPM provided by Microsoft, runs in trustworthy hardware, ARM TrustZone. In which, the TrustZone secures the execution of the TPM and separates such a sensitive component from the applications and most of the system.

4.1 Related Work

The notion of theoretically using TPM for mobile phones is not new. However, there is no TPM chip available in mobile platforms. Several pieces of research adopted the TPM to preserve privacy and provide trust in the system. Some approaches [142, 146, 148] used mini TPM emulator as a Linux Kernel Module (LKM) and is part of the Android kernel. In [146], Nauman and colleagues present an attestation

mechanism for VM-based architecture OS, Android, and show the feasibility in terms of complexity and battery consumption. In subsequent work by the same group [142], a protocol for keystroke dynamics analysis is proposed, which enables web-based applications to make use of remote attestation in Android platform.

However, these approaches relay on an emulated TPM that is part of the Android kernel, thus, the trustworthiness of this element is questionable. Such an approach does not get the benefits of a TPM being a secure element, which is protected from the OS and other code. Also, there is nothing mentioned about the execution environment and the TCB of the same environment. The TPM is a good candidate that provides much functionality for securing a system. However, once compromised there is no guarantee that it can still assure the confidentiality of the data or the integrity of the system. Hence, a better approach to take advantage of its full functionality would be to use a hardware TPM or isolate its execution from the rest of the system.

Other papers [145, 149], extended the OS kernel to assure that a trusted code runs before the rest of the application, thus enables the verification of integrity before passing the execution to applications in the same system. GuarDroid [145], provides a trusted path between the user and the app server by leveraging a service added to the OS. The user inserts his/her password to a trusted input provided by the OS, which is in turn encrypted with a protected secret key. The aim is to hide passwords from untrusted apps while preserving external behaviour. In [149], Zhang el al. develop a system to ensure that a secure kernel is booted. Once the kernel is booted, the work is done by the measured software to ensure a well-behaved execution of the system

through an agent that enforces authenticated security policies. These approaches assume trusted OS or other measured software that runs in the same environment as the OS. Yet, the user is not protected from vulnerabilities in the code. Once the OS is compromised there is no assurance that the confidentiality of the data is kept. To overcome the shortcomings of previous work, we adopted a different approach in which we promote the use of a stand-alone element, the TPM, that is not an extension of the OS. We develop an application that makes use of software TPM that runs in a hardware-based secure environment, ARM TrustZone. We show how a TPM can elevate security that is also usable in mobile platforms by requesting minimal involvement from the user.

4.2 Adversary Model

In this work, the focus is to protect the confidentiality and integrity of secrets (e.g., keys, passwords, configuration values) from vulnerabilities in the OS and applications. To achieve this, the functions executing on those secrets must be protected from tampering. The adversary is able to exploit vulnerabilities in the system to tamper with the cryptographic function execution or gain access to brute force his way into privileged operations. We assume that the adversary can not place himself as man-in-the-middle, however, has access to the system's memory and is able to call any function of the TPM Base Services (TBS), either through the OS or an application when a vulnerability is exposed. The adversary is also able to tamper with the

execution of functions and change the application's heap and stack data in memory by exploiting a buffer overflow vulnerability.

4.3 Framework for Secure Authentication Protocol

We propose a framework that uses trusted computing technology, the TPM to secure sensitive data such as banking information in the context of mobile devices. The Local Mobile Application (LMA) uses the TPM to store the identity and the credentials used for the user's authentication to the remote server. The local mobile application authenticates the user's identity in each access using a PIN or a fingerprint. The PIN is an important component for using the LMA, as it is used to access the application, in addition to being used as a configuration parameter for local TPM operations. The LMA establishes a trusted channel connection to the remote application server, and it undertakes mutual authentication with the bank server.

To focus on the strength of using TPM in a smartphone, we assume the following: (1) the operating system is root verified and each application runs in a different sandbox. In addition, (2) a trusted code manages the dynamic Platform Configuration Registers (PCR) when moving between applications by storing the PCRs value when another application is running and retrieves them when needed. We argue that these assumptions are reasonable in the context of mobile platforms since an attacker cannot

run modified operating system without breaking the secure boot, hence, an adversary can only exploit vulnerabilities in the design to escalate their privileges both in use-space and application. Our framework consists of two parts, the registration process and the login process both of which use a trusted channel to communicate between the device and the bank service. The channel carries all requests and responses between the user's device and the bank server through the secure Transport Layer Security (TLS) protocol connection to protect the exchanged data and authenticate the bank server. The bank maintains a white list of the acceptable entries which is obtained during the initial registration to the bank service: each entry is tied to a user's data and used by the bank for user authentication. The registration and login processes are explained in more detail in the following sections.

We use a sequence of authentication and attestation processes between the user, the mobile application, and the bank service. The user authenticates to the mobile application and only then the app establishes a SSL/TLS session that authenticates the bank service, which then authenticates the user in two steps, and attests the mobile application. In the first authentication process, the activation key plays a major role in authenticating and associating the user with the app since the activation key is unique per user. This is important because only a certified app with appropriate activation key can authenticate to the bank service, thus, protecting the users against phishing attacks. In the second step, the app provides the user's credentials which are sent to the bank service to get access to the user's account. Both steps are essential in the initialisation process because the activation key is not associated to any user,

and the credentials associate the activation key to a user.

4.3.1 Mobile application registration

Figure 4.1 describes the process of registering a new account with the mobile application. Applications can be downloaded from the Application (App) store of the platform provider: only signed and certified applications that have a unique license should be used. The license validity is verified by the OS before installing any app on the device. During boot time the kernel ensures the OS OS measures the software and extends the measurement into a dynamic PCR. We assume that the trusted operating system uses application-specific PCRs and manages them according to the running application thus allowing us to use the dynamic PCR without worrying that other applications might overwrite it. This gives control to the bank service, enabling it to revoke a configuration from its white list when a version is out of date or contains security holes. This method enables the bank to protect its users when discovering any security vulnerability in the application they provide. For initialisation, an activation key is needed to unlock the app: this is provided to the user by the bank. It is used to associate an application to the user and allows stronger authentication with every access. The activation key is hashed and extended into the dynamic PCRs, and stored safely using a Storage Key (SK) derived from SRK created by the TPM. Using the TPM to verify the user's identity mitigates software attacks, while keeping the integrity and confidentiality of the user's data. Using a dedicated key, fingerprint or password (e.g., PIN), the user's identity is corroborated to the application at every

use. This key is hashed by the TPM and secured using a SK which associates the user's key to the platform. The application verifies the authenticity of the user's key with every start/resume of the application. The LMA establishes a trusted channel with the remote application using TLS to enable secure communication between the mobile and the bank server. The application requests the account details in order to access it from the remote application, which in turn authenticates the user's details and creates an access certificate for future access must be signed. The application stores these results in the TPM once, and uses them as OpenID to get the data for the requesting service when working with two-step authentication. A blob with the user's credentials, associated to the relevant dynamic PCRs is created and stored in the device. The dynamic PCR state is signed using an Attestation Idendity Key (AIK) and sent to the bank for user attestation with every connection.

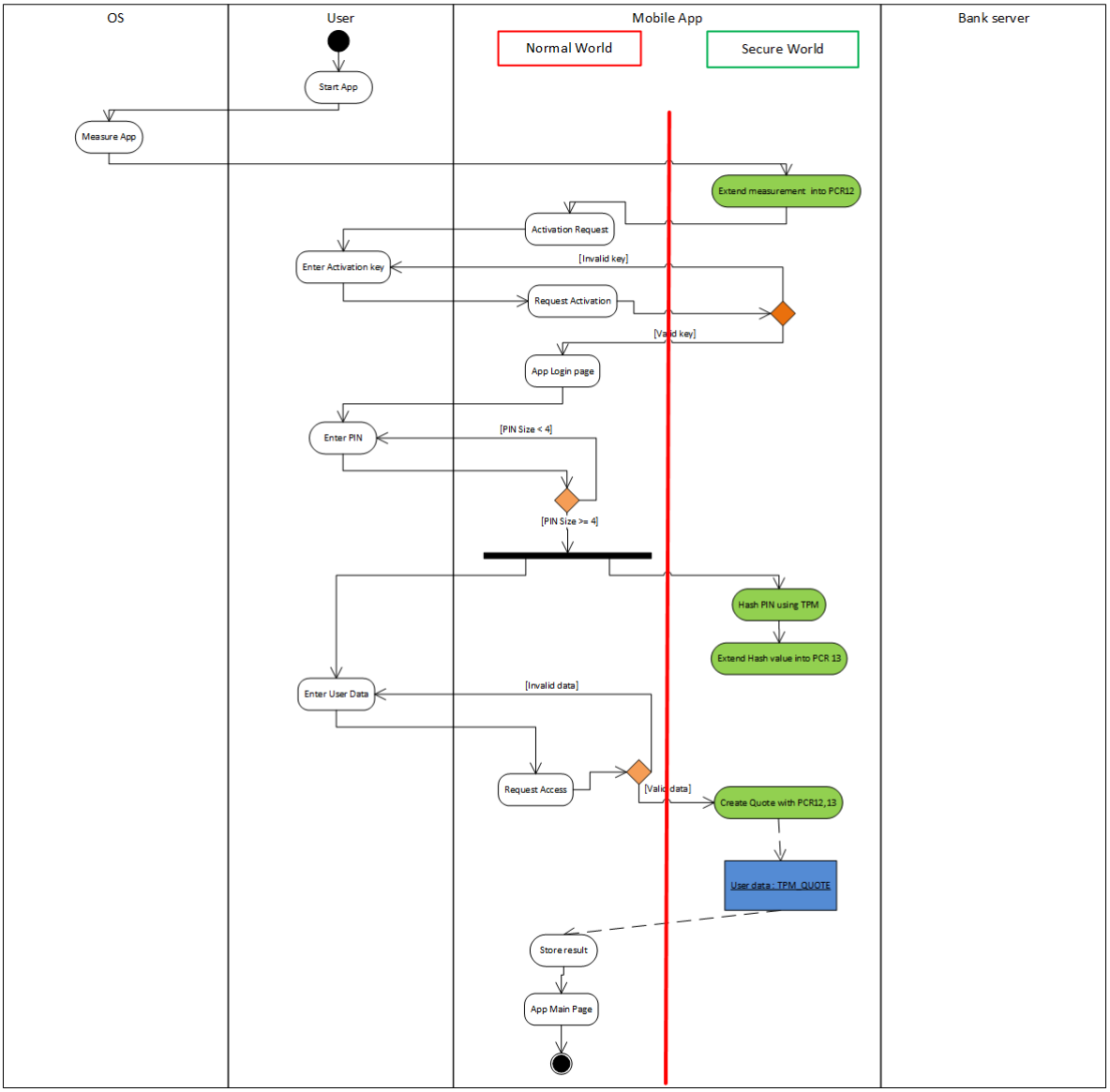

Figure 4.1: User Registration Flow: Describes the execution flow on Windows Phone with four parties involved. The registration flow takes advantage of ARM TrustZone to secure sensitive Trusted Computing operations, such as tampering with the PCR, or hashing functions, and most importantly producing a quote that is used for attestation. From the perspective of execution, each one of these commands is considered atomic operation since none of the software in the *normal world* can tamper with its execution.

4.3.2 Secure mobile login

Figure 4.2 describes the sequence of events in using the app after completing the registration process. In order to establish a connection with the bank, it is important to attest and authenticate the mobile application and the user's data stored within it against the bank's records. During boot time, the OS measures the application and extends the value into the dynamic PCR, which then passes the control to the banking application. The app requests a PIN for user authentication, and it uses the TPM to hash the PIN before extending it to the dynamic PCRs. In order to communicate with the bank server, the device needs to attest and authenticate its identity. The app extracts the user's data through TPM unseal operation on the stored blob, the success of the operation requires the right values of the dynamic PCRs. The three conditions that need to be met for a successful unsealing operation are right values for (1) the activation key, (2) the hashed software, and (3) the user's PIN. These PCR values reflect the state of the device; they are signed using AIK and sent to the bank server to complete the client's attestation and authentication, for establishing a trusted channel. The bank authenticates the user's device PCR values against a list of known "good values" [150]. On completion of the attestation process, the user's credentials are sent automatically by the mobile device to the bank server for authentication. The bank verifies the user's data and grants access to the user. The user's credentials are used for first time accessing the bank services and on later access the two-step authentication provides two independent pieces of authenticity.

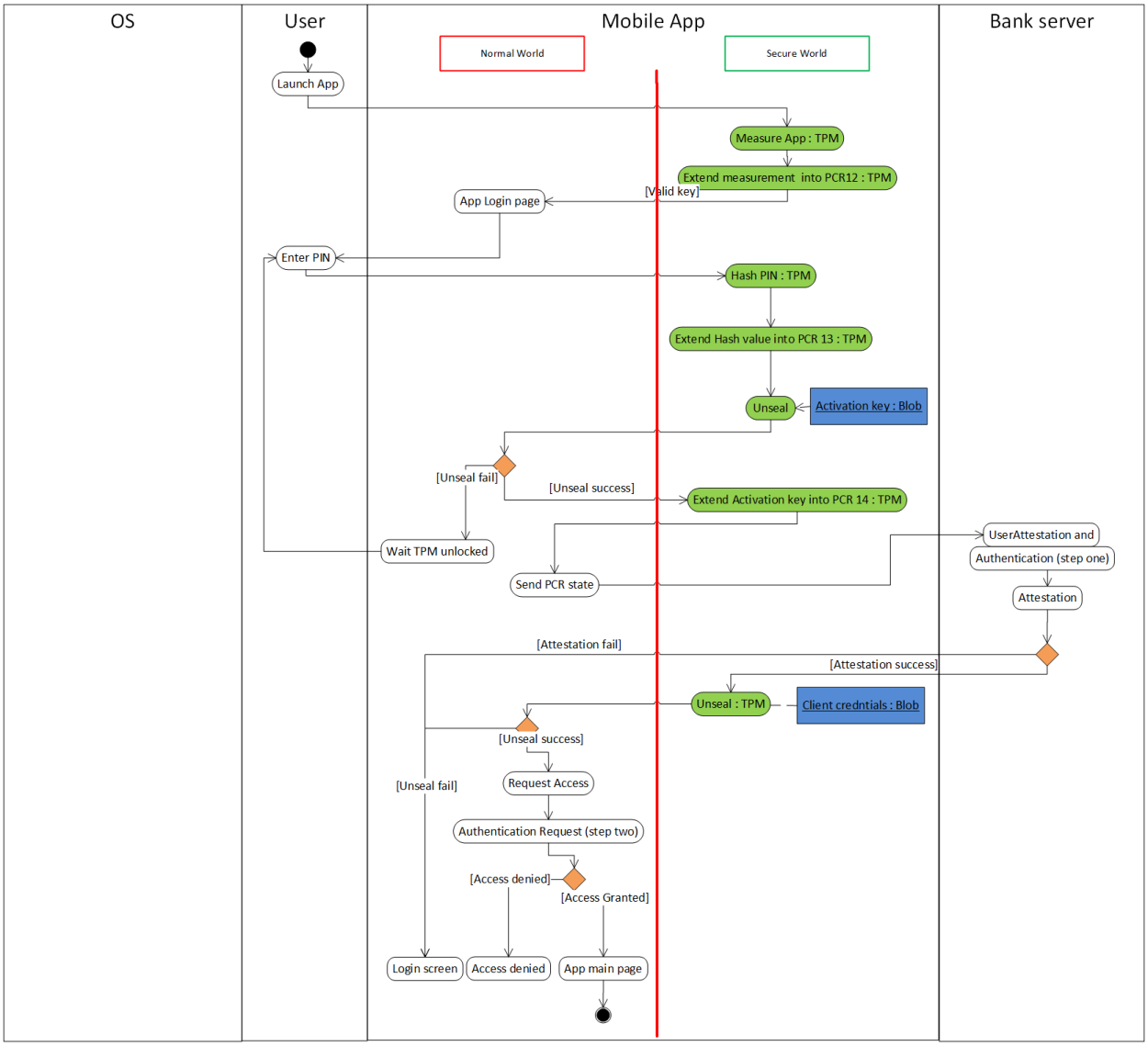

Figure 4.2: User Login Flow: the user login is considered a critical operation since most attacks happen during run-time. Thus, securing each operation is critical. In particular, an adversary would have a strong control over the platform if he is able to produce one of the results produced by the software in the ARM TrusZone (e.g. unsealing operation or producing a blob).

4.4 Proof of Concept

In this section we describe the implementation of the application's architecture described in the previous sections. The objective of this implementation is to demonstrate that using current technologies, it is realisable to achieve the security objectives we specified.

4.4.1 Implementation Specifications

We implement an application for Windows Phone 8.1, whose architecture is based on Windows NT kernel, the same kernel used for Windows 8. ".Net" can be used for developing applications for both operating systems, and the implementation used for developing an app in one environment, can be suitable for the other environments. We designed the application for Windows Phone 8.1. We use C# to develop a proof-of-concept application that uses the software TPM in a smartphone. The application communicates with TPM2.0 through Windows TBS system call in C#. In order to use the TPM we use the TBS function that receives a "TPM command" in binary format as an input to the TPM. In our implementation we couldn't directly use the TSS.Net library [151] since it is not compatible with Windows phone environment, thus, we had to modify it to render it compatible. Notwithstanding, in the near future these libraries should be able to support Windows phone environment since the base code, as introduced by the Trusted Computing Group (TCG) specification for communicating with the TPM, is in C. Our system consists of two devices and

a channel to communicate between them. The first runs the bank application (the client) and the second runs the bank services. The communication channel between the two ends uses the SSL protocol for communication, and we use the SSL library developed by Microsoft to make use of this protocol.

4.4.2 TSS.Net

The TSS.Net library is a managed code (dll) written in C#. The library is open source and is used for easy access to the TPM. It makes it easier writing Windows applications using the TPM. The TSS.Net handles all the low level functionality when using the TPM, hiding all the complex interactions from the developer. In addition, the library communicates with the TPM simulator through a network socket TCP/IP to give the opportunity of development and debugging.

4.4.3 TPM2.0

The TPM2.0 simulator was developed by Microsoft and the TCG based on the TPM 2.0 specification [74]. It emulates the TPM 2.0 commands in software. To our knowledge there is no hardware TPM aligned to TPM 2.0 specification. We use the TPM 2.0 simulator as a tool to test our framework. Microsoft surface, and Nokia Lumia 830 include a TPM 2.0 simulator running in the TEE of Arm TrustZone [67] processor to protect its secrets and execution.

4.4.4 Interface Between The *Secure World* and *Normal World*

The applications interact with the TPM through the TPM interface as defined by TCG. The implementation of the former software interface is within the TSS.Net and TBS libraries that facilitate communication between the application running on top of the OS and the TPM. In turn, the TBS acts a translator of application requests to binary commands in the same way the TPM chip receives commands. The TBS submits binary commands which are received by the TEE Monitor and directed to the TPM in the *secure world*. Figure 4.3 describes the software architecture of windows phone 8 and the services that felicitate the communication between applications and the TPM.

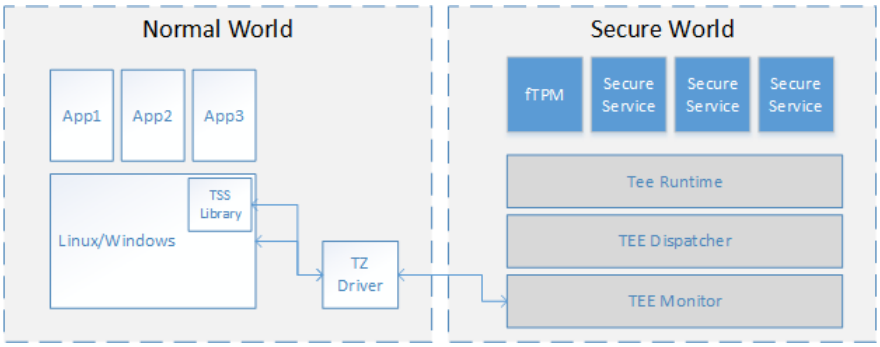

Figure 4.3: Windows Phone 8 Software Architecture

4.5 Evaluation

The goal of our design is to provide a framework for mobile applications which is easy to use and at the same time does not compromise security. In order for users to access their accounts, they are required only to authenticate themselves to the

mobile application by providing a PIN of their choice. The initialisation process is the most involved process as the user needs to provide multiple parameters such as the activation code for the application, gating PIN to the application, and user's credentials to access the bank services. All this information is initially obtained from the bank. It is worth noting though, that this procedure is performed only once during the setup time and the user is relieved from providing these data with future access to the bank account by solely using a PIN based login system. In order to provide a secure end-to-end solution we recommend the use of a trusted computing technology, the TPM to protect the integrity and confidentiality of the user's data, SSL for secure channel connection, and two level authentications. The first level authentication in the login process can be done with the attestation data only, this association between the authentication and attestation in the setup process enables the user to replace bank credentials with attestation data for access control to the bank services. The use of these security elements are hidden from the user making it easier to use, and at the same time provides secure connection to the bank services.

4.5.1 System benchmarks

The proof of concept implementation was tested on a Nokia Lumia 830 with an ARM Quad-Core Cortex-A7 CPU running at a clock frequency of 1.2 GHz and a TPM2 simulator (version 1.0) running in the *secure world* of ARM TrustZone. Using our implementation we are able to evaluate the time scale and relative time costs for the different operations performed. Table 4.1 shows the time measurements of the

Table 4.1: Table 1. Time of execution measurements.

Operation	**Time (ms)**	**[SD]**
RNG	0.896	[0.17]
PCR Read	1.05	[0.12]
Sha1 Data Hash	1.06	[0.15]
Sha1 Key Sign	0.4	[0.07]
Extend PCR	1.2	[0.11]

operations performed on a software TPM 2.0 that runs in the secure zone of an ARM TrustZone processor. In order to test the performance of these operations in Nokia Lumia 830, we execute each operation 10,000 times and assume equal probability for each measurement to calculate the Standard Deviation (SD). It can be seen from Table 4.1 that the operations used do not affect the user's experience as the time of the execution is merely milli-seconds. In addition, the max standard deviation is 0.17 showing that even with deviation the time of the execution of each of the operations is still negligible. Table 4.2 shows the differences between a credential approach compared to the several stages we suggested in our framework. We determined the credentials' size based on the banking application. Our initialisation process uses more data compared to the credentials approach. However, in steady state we use less data when working with one step authentication and more data when working with two steps. The two step authentication adds another level of security where the bank checks the user's credentials with every access request by the user, while the one step uses the remote attestation and application Activation Key only.

Table 4.3 summarises the number of operations used in the registration and the login processes respectively. In the login process, one unsealing operation could be spared when giving up the second authentication stage.

Table 4.2: Number of bytes used in the two approaches

	Step	**Number of Bytes (bytes)**
Approach 1	Credential based	44
Approach 2	Registration process	84
	Login (1 Step)	40
	Login (2 Steps)	84

Table 4.3: Number of operations in the Registration and login process

Operation	Number of calls (Login Process)	Number of calls (Registration Process)
Sealing	0	2
Hashing	1	1
Unsealing	1-2	0
AIK Generation	1	1
Extend PCR	3	3

Finally, Table 4.4 summarises the points addressed in this paper compared to existing technology. We mention how the current technology addresses each point, the flaws that exist in current technology, and how does our framework overcome them. It can be seen from Table 4.4 that the proposed solution relies on trusted computing rather than certificates and external devices, thus, making the solution more usable since it does not require additional devices. In addition, the proposed solution does not rely on certificates which are considered to be unreliable and requires trusting all certification authorities in the chain. In contrast, attesting the software of the server provides assurances about the status of the software used by the server, hence, a client can choose to reject the connection if server reports an unexpected software.

Table 4.4: Mobile Application summary compared to current technology

	Current Solution	Vulnerability/Issue	Our Solution
Brute Force Attack	External device Password Generator	Requires additional device	TPM anti-hammering
Man-in-the-Middle	SSL/TLS	Certificates managing handling	Dual Authentication & Remote Attestation

4.6 Security Evaluation

Our work proved to be effective against vulnerabilities in the kernel for which we exploited a vulnerability in Windows NT by gaining privileged access to the system. In an attempt to brute force the PIN of the user, the TPM running in the *secure world* employed the anti-hammering and blocked attempts for 5 minutes after 10 wrong attempts. The execution of the TPM function was secured as the exploit was in the *normal world*, thus, had no effect on the execution in the *secure world*. However, denial-of-service was possible after injecting code and changing the implementation of the TSS library. The secrets used by the TPM for sealing the PCR were kept secret in the *secure world* and access to it was allowed through dedicated interface. Every other attempt of submitting different commands caused denial-of-service and caused the TPM to lock for several minutes.

4.7 Discussion

In modern operating systems (e.g., Linux, Android, iOS, and Windows Phone), the OS enforces permission based access control to secure an element such as a file, a

resource, or a service. This approach is proving to be inefficient as the number of vulnerabilities in OS is rising. The use of software TPM allows the use of standards-based security that is also flexible to adapt in several systems such as smartphones and cloud environment. In particular, it is important to take into consideration the attack vector that rises in a system from adopting such an approach. The approach in smartphones for using software TPM is running such sensitive element in a secure environment that is isolated from the applications and the OS, such as ARM TrustZone. This environment is referred to as the *secure world*, and it's protected from applications that run in the *non-secure world* or *normal world*. However, the software TPM is not the only code executing in the *secure world* and others, such as parts of the OS and the kernel, also run in this environment. Hence, the software TPM is not protected from vulnerabilities in the rest of the code that is running in the *secure world*. In the aforementioned approach, the TPM is a stand-alone element from the OS and the kernel, and runs in a trusted secure environment. However, it cannot be considered as secure as a hardware TPM which is not exposed to such vulnerabilities. In addition, an important consideration is the interface between the *secure* and *normal worlds* since a strong adversary can place himself as man-in-the-midle between the application and the software TPM. As mentioned in the previous chapter, a mindful partition should be wary of the information flow between trusted and untrusted environments. However, even such an attack will not be able to reveal secrests.

Furthermore, in the design described in this chapter there is no assurance that the

user's input is not spoofed or modified by an untrusted app or the OS. The software TPM when running in a trusted environment can protect against brute force attack of passwords/PINs using anti-hammering, according to which, the TPM locks itself after a configurable number of wrong attempts. This prevents the attacker from guessing the password in a reasonable time. However, passwords can be sniffed by untrusted applications through the keyboard service and the OS; for the sake of this discussion we assume trusted channel. As a result, spoofed data can be used by an untrusted app to access sensitive information and authenticate itself to the TPM and perform any operation like a valid user.

4.8 Summary

Authentication protocols are being used on mobile phones to authenticate users to external services. Many approaches have been suggested to secure the execution of sensitive operations. Many papers addressing security in mobile applications rely on old technologies and suggest solutions within their limitations. However, many of these methods are point solutions and are not easy to use, which presents a serious limitation since the user becomes discouraged by the many layers of security measures introduced to bypass them. Many methods aim to enhance the security of applications, by requiring more user input like copying codes from a smart card or SMS message, putting more responsibility on the users to enhance security. These methods became impractical and inconvenient to use, and having simple access control

while not compromising security becomes essential. The use of a PIN is an easy access control for mobile applications and hides all the security levels from the user making it very convenient and simple to use without compromising the security standards. In this chapter we show how using TPM 2.0 in trusted and untampered execution environment can mitigate some of the existing threats on mobile applications, to improve current security schemes. We propose a novel communication protocol that uses both authentication and attestation for setting up the user's account, hence removing the pressure of securing the credentials from the user alone. In addition, we point to the weaknesses in using this technology. The prototype implementation demonstrates feasibility and a path to an implementation on a mobile platform with the necessary trust characteristics and reasonable performance.

If someone else can run arbitrary code on your computer, it's not YOUR computer any more

Rich Kulawiec

5

Application Partitioning for Mitigating Software Vulnerabilities

This chapter is based on the following publications:

- A. Atamli-Reineh, & A. P. Martin, "Securing Application with Software Partitioning: A Case Study using SGX". *In Proceedings of the 11th EAI International Conference on Security and Privacy in Communication Networks (SECURECOMM'15)*, 2015 [4].
- A. Atamli-Reineh, R. Borgaonkar, R. Balisane, G. Petracca, & A. P. Martin, "Analysis of Trusted Execution Environment usage in Samsung KNOX". *In Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*, 2016 [2].

Contents

5.1 Background and Related Work

5.1.1 Software Partitioning

Privilege separation of applications into trusted and untrusted code is a well-established technique that is used in many modern systems. One of the most common examples is the separation of the Operating System (OS) kernel from the applications running on top of the OS. Although this approach is already in use, it has arguably not yet reached its full potential. That is due to the absence of suitable technologies to facilitate privilege separation. In light of this shortcoming, new technologies are still being developed to give support for privilege separation, not only in software but also in hardware. This includes tools that assist with the partitioning of software, as well as technologies for strengthening the isolation between different partitions.

The principle of least privilege [72] limits the harm from a malicious injected code as well as prevents exploits of bugs from accidentally leaking information. However, very few programmers actually divide their code into minimally privileged compartments. One potential reasoning is that the OS has no proper isolation primitives offered to applications, and are hence cumbersome when an attempt is made to limit privilege.

The use of processes as containers is questionable as illustrated by considering the fork-system call. In this case, sensitive data must be scrubbed from memory to prevent a child process from gaining unauthorised access. This cannot be an OS

primitive and hence requires a larger involvement from the programmer. In many cases this might prove not sufficient enough as it is usually impractical and even impossible to consider all potential data that might require scrubbing.

5.1.2 Related Work

5.1.2.1 TCB Minimisation

The size of the Trusted Code Base (TCB) in systems and applications is a topic of vast interest in research. This section introduces TCB minimisation, a topic which has been discussed in commodity systems [152] by McCune et al., however, not in applications. In particular, the topic of executing sensitive code in isolated and trusted environment has caught the attention of many researchers. McCune et al. presented Flicker [153], an infrastructure for code execution in isolated and trusted environment. In their work they rely merely on 250 Lines of Code (LOC) in the TCB to provide strong isolation. For the most part, they appreciate that 250 LOC is a tiny code, therefore formal assurance of its execution is more trusted since it is feasible to verify the code. Nonetheless, an application running in an isolated execution environment can be thousands of LOC and isolation between several parts in the application space is essential to prevent exploits by unfortunate vulnerabilities. The same group presented TrustVisor [62] a pointed purpose hypervisor that provides code and data integrity and secrecy for sensitive portions of an application. TrustVisor provides application developers with an isolated environment for code execution and

data storage on untrusted platforms. Moreover, they argue that it is easier to formally verify a small TCB code, thus, making it more trusted when executing in a Trusted Execution Environment (TEE) .

Another research effort that takes a similar approach is that of Singaravelu et al. [154] who showed that reducing TCB complexity can result in enhancing the security of the sensitive part of the application. The sensitive part is executed in a process called AppCore while the rest of the application is executed on a virtualised untrusted operating system. This approach is supported by three real world case-study applications.

Murray et al. suggest the use of dynamic libraries, commonly used in software development, as a mean for disaggregation [155]. In this work they propose dividing the software using an existing development technique to one or more dynamic libraries. In turn, a privilege software such as the kernel or the Virtual Machine Monitor (VMM) enforces isolation between the two domains, the host process and the shadow process that includes the dynamic libraries. Their approach uses the Xen hypervisor for coarse-grained isolation between the two domains, this classifies as scheme type 2 in our proposed framework presented in chapter 6.

5.1.2.2 Detection and Monitoring

An alternative method to using isolation to protect secrets and mitigate the exploitation of vulnerabilities is through monitoring the execution of the software.

Geneiatakos and colleges suggest using virtual application partitioning to dynamically adapt the software defences based on the current execution partition of the application [112]. The main focus of their work is identifying authentication points and "sensitive" data in binary applications using techniques such as Instruction-set randomisation (ISR) and Dynamic Taint Analysis (DTA) on the different partitions of the application. For example, for data that is classified as *important* it is desirable to adopt strict policy or tracking information flow to prevent auditing or leakage of this information over the network.

In the web application domain, Prokhorenko et al. expose the shortcomings in many web protection techniques [48]. In their work they survey and systematise existing protection techniques, and discuss the limitations of existing methods. Their results show that technical errors in applications do exist, and that imprecision in protection techniques such as sanitisation verification, detection, and manual reviews can be used, but are unlikely to detect all technical errors and faults. The findings of their work are very relevant to our context, since imprecision in techniques was the main driver for our work. We believe that there is a profound need for another layer of protection, such as the one proposed in this work, to mitigate the impact of exploits. In another publication from the same group [156], an application protection model is proposed in order to facilitate the implementation of universal application protection to counter attacks. This shows an alternative view of injection attacks and unifies different types of injection attacks. This constitutes one of the backbones of our future work towards mitigating injection attacks using TEEs.

5.1.2.3 Sandboxing

Sandboxing is a well-known container pattern in which the execution is isolated from the rest of the system.To elaborate further, the isolated software cannot interfere with the execution of external software. Many pieces of research proposed Sandboxing to achieve security and privacy. In Robusta [157], the authors propose a framework that provides security to native code in Java applications. Robusta, isolates native code and prevents external modification of the data and keeps the data's confidentiality. In order to achieve this isolation, the implementation has to change parts of the OpenJDK. The Sandboxing is achieved through separating the address space into two spaces, non-writable code region, and a non-executable data region. The Robusta uses hooks in the Java Virtual Machine (JVM) to intermediate between the JVM and the native code.

In another work, Dune: safe user-level access to privileged CPU features [158], the authors present a design that exposes the CPU virtualisation feature of Intel CPU to user applications. Instead of using the virtualisation for Virtual Machine (VM) as used in many systems, they use virtualisaion for a process which is much lighter-weight than a VM, hence, enabling the support of large number of processes. Dune's main changes are on the way it leverages the hardware protection rings and hyper calls. Dune leverages the protection rings for processes, the privileged rings are exposed to normal processes which allow a process to run in a ring 0 privilege. Furthermore, instead of using system calls it uses hyper calls.

In the mobile devices domain, the hardware architecture of Arm TrustZone performs separation between the "secure world" and the "normal world". However, it is very unlikely to get the approval of the Original Equipment Manufacturer (OEM) to get a new secure code installed in the secure world. That's mainly to keep the TCB inside the secure world as small as possible. In TrustICE [159], the authors ensure isolation of a secure code in the normal world, called by the authors *Isolated Computing Environment (ICE)*, through using Trusted Domain Controller (TDC) that resides in the secure world. To establish an ICE, a request is sent from the normal world and handled by the TDC, which in turn saves the status of the executing software (the Rich OS), configures the registers to prevent handling interrupts, and after verifying the code integrity of the ICE it is saved in a secure memory. The entire switching between the Rich OS and the ICEs is done using the TDC to keep the isolation between the two. Other proposals use software approaches to protect users' data, either by expanding existing parts of the system or adding dedicated code to enforce access control [160, 161, 162, 163]. These approaches are orthogonal to our approach and may rely on our approach to mitigate information leakage vulnerabilities. In our work, we rely on TEE s to reduce the likelihood of software vulnerabilities that can be exploited by an adversary, as well as reducing their impact through partitioning. For instance, the authors [160, 161, 164, 165] explain attacks to obtain users' data and recognise that these attacks wouldn't have been possible if the data were encrypted. Our partitioning approach using TEE s provides an efficient solution to this challenge.

In the network application domain, Kim et al. use OpenSGX, an emulator of

Software Guard Extension (SGX) to solve the privacy and security issues in Software Defined Network (SDN) based inter domain routing and peer-to-peer anonymity networks (Tor) [166]. In SDN based inter domain routing a common approach is to use secure multiparty computation, using SGX to verify the integrity of the code performing the computation at each node, can solve this issue. A quoting enclave at each node sends a report of the computing code and data, which is verified by the receiving quoting enclave at another node. In addition to the privacy issue with SDN, Tor relies on a volunteering node that can easily modify the software. To solve this shortcoming in Tor, the authors ported Tor source code into an enclave, which in turn can be verified against signed certificates. Such combination ensures the integrity of the code executing inside an enclave to be certified.

In Security-Oriented Analysis of Application Programs (SOAAP) [35], the authors present a tool for evaluating proposed compartmentalisation patterns. The tool uses code-annotation to define the sandboxing boundaries of the compartment's code. For example, it defines rules on sensitive data such as keys, and take into account past vulnerabilities as measure to the effectiveness of the applied compartmentalisation. In SOAAP, the authors use compartmentalisation patterns [167] to control the delegated rights of a compartment. For instance, a compartment that handles untrustworthy code or data with a risk of containing code execution vulnerabilities delegated minimal privileges to mitigate exploitation of vulnerabilities.

Kilpatrick et al. [71] propose a library that simplifies partitioning applications for privileged UNIX daemons. Privman requires an application to be separated into

two processes: the privilege process as the trusted process and the main application as the untrusted process. The privilege process preforms privileged operations on behalf of the main application and it limits the privileges of the main application by enforcing static configurable policy access. The privilege process can limit access to files, bending ports, and other privileged operations.

In [168] Strackx proposed Fides- a security architecture that consists of two parts: a run-time security architecture and a compiler. The run-time security architecture is based on memory access control to protect applications. The modules are divided into a private section, where sensitive data is protected and accessed by the relevant module through limited interface, and a public section that contains the module's code. The second part is the compiler which is responsible for compiling standard C code into protected modules. In another work [169], Cheng et al. presented Driver-Guard, a hypervisor protection mechanism to shield I/O flow from a malicious kernel. DriverGuard protects a tiny fraction of the code that is sensitive, such as biometric authentication. However, they assume secure boot-up and load-time attestation to ensure the hypervisor's security in the bootstrapping phase.

In [170] Li et al. introduce MiniBox, a two-way sandbox that isolates the memory space between OS protection modules and applications. Unlike most approaches it aims to protect the OS from untrusted applications, but also protects the applications from a malicious OS. In Minibox, the authors focus on the two-way Sandboxing and don't address the porting efforts for legacy code, and suffice by mentioning that the porting efforts are similar to the porting effort on NaCl [171].

In [172] Vasiliadis et al. introduce PixelVault, a system that uses GPUs to secure cryptographic keys. In PixelVault the private key is created inside the GPU and never leaves or leaks it even in the presence of malicious OS. However, this is limited to the private key since PixelVault can not use the GPU to secure keys negotiated at run-time such as the session key or key pairs. Thus, malicious software can act as a man-in-the-middle.

Liu et al. propose a mechanism for automatically partitioning applications for security, using static and dynamic analysis. However, they don't specifically consider the different ways in which applications may be partitioned, as we have done in this chapter [173].

Partitioning privileges between hardware and software is not a new paradigm [174]. Hardware/Software partitioning has shown improvement in performance, energy consumption, and optimised run-time.

5.2 Adversary Model

The adversary model defined in this chapter is the adversary model used for all use-cases mentioned in this thesis. At the platform level an adversary has the capability to run arbitrary software on the platform, read all platform memory, and manipulate the OS (including booting another OS). The adversary model is inspired by previous work [175, 176]. Specifically, the capabilities of the adversary are as follows:

1. `Install` allows an attacker to install and run new software on the platform including: a new OS, other applications or monitoring tools, or arbitrary code.

2. `Read` allows an attacker to read data from the platform's physical memory or persistent storage.

3. `Intercept` allows an attacker to intercept the communication between components and data in transit, such as temporary memory data used by the application.

4. `Delete/Modify` allows an attacker to modify and/or delete data from memory or persistent storage, or roll-back memory to an earlier state.

5. `Inject` allows an attacker to inject code to external software e.g. the OS or other software that may be vulnerable to code injection in order to obtain memory data or root access.

6. `Corrupt` allows an attacker to corrupt data/meta-data of physical memory and persistent storage.

7. `Exploit` allows an attacker to exploit vulnerabilities in the OS and applications to obtain sensitive information or temporary data used by an application. This also allows the adversary to exploit any available side-channel attacks against the TEE (e.g. [177]).

This is a realistic representation of an adversary who has been able to gain root access to a system (e.g. through malware or exploiting a vulnerability in the OS)[178,

179, 175, 176]. However, the adversary's targets are primarily the applications running on the system, since these represent the greatest value to the adversary. For example, these applications could contain users' access credentials or financial information, which the adversary could abuse or sell, as well as system secrets such as private keys, which the adversary could monetise through subsequent attacks. Therefore, the adversary aims to obtain these pieces of sensitive data from the applications, either directly or by exploiting vulnerabilities in the application. However, it is assumed that the adversary cannot compromise the TEE provided by the platform. Whilst this is hypothetically possible, the cost is widely judged to be very high, and such attacks almost certainly require physical access to the platform. This class of attacks is out of scope for our present study.

As explained in Chapter 2.1, a TEE isolates the execution of specific pieces of code from the rest of the system, which may be untrusted. This prevents the untrusted platform from modifying the isolated code, and thus protects the integrity of this code. Furthermore, the TEE can store data within this isolated environment. This protects the confidentiality and the integrity of the data since the data cannot be accessed or modified by the untrusted platform. Therefore, the primary functionality of a TEE is to protect the integrity of specific pieces of code, and to protect the integrity and confidentiality of specific pieces of data from an adversary who potentially controls the host platform. However, side-channel attacks are beyond the scope of this thesis. This chapter focuses on approaches using TEEs to mitigate software vulnerabilities.

5.3 Security Objectives

Having defined the adversary model for the use-cases in the previous section, this section defines the security objectives of this thesis. From the perspective of an application running on untrusted platform, the TEE functionality can be used to protect security-sensitive code and data. In this context, the following definitions are used:

- **Security-sensitive data:** any data for which the security of the application depends on the *confidentiality* and/or *integrity* of the data.
- **Security-sensitive code:** any code for which the security of the application depends on the *integrity* of the code. It should be noted that *confidentiality* of code is beyond the scope of this work, since this cannot be achieved by current hardware-based isolation technologies (e.g. SGX does not have this capability).

It follows from the above definitions that any code that directly handles security-sensitive data is itself security-sensitive code. If this were not the case, the adversary would be able to modify the code to reveal/modify the security-sensitive data, thus subverting the security requirements. Since we are dealing with the security of applications on a platform, we consider two categories of security-sensitive data: *data in use* and *data at rest*. Data that is communicated over a network is beyond the scope of this work, and thus we do not consider *data in transit*. In the context of a Transport Layer Security (TLS) library, an example of data in use would be a ses-

sion key established during the TLS handshake phase, whereas an example of data at rest would be the long-term TLS private key, while it is not being used by the application. The security objectives described are therefore to ensure the *integrity* of security-sensitive code and the *confidentiality* and *integrity* of security-sensitive data, with respect to the adversary defined above.

In general, it can be argued that for a given application, security-sensitive code and data constitute a relatively small fraction of the code and data that make up the application. Isolating these sensitive components from the rest of the application protects them from any vulnerabilities that may be present in the rest of the application. Therefore decreasing the likelihood of the adversary being able to compromise the security objectives of the system. It has been shown that hardware-assisted partitioning technology, such as Intel SGX, can be used to achieve this type of isolation [180, 181].

Given the powerful adversary model described above, it is arguably not possible to provide availability guarantees, since the adversary has a very high degree of control over the system. Even if the security-sensitive code is run in an isolated execution environment, the adversary can run arbitrary software on the main platform, leading to resource starvation for the security-sensitive code, or simply to halt the main platform. For these same reasons, technologies such as SGX also do not aim to ensure availability of security-sensitive code or data.

5.4 Reducing The Impact of Software Vulnerabilities

This section explains how leveraging TEE as a container can reduce the impact of software vulnerabilities. TEE implemented under the same processor has several properties and capabilities. For instance, a TEE technology- Intel SGX protects enclaves data when residing in memory by encrypting its memory pages in memory and decrypting data only when it enters the CPU. Thus, such technology protects the integrity of the code and confidentiality of the data from unauthorised access by external software (e.g. exploit of code in the same process but not inside an enclave). Built on these two properties and the requirements in sections 5.4,5.6 and 5.5, a metric of potential types of software vulnerabilities is defined and examined.

5.4.1 Vulnerabilities

We start by listing several vulnerabilities and explain how can they benefit from the use of the TEE .

5.4.1.1 Buffer Over-Flows

Also known as buffer over-run, is when a program read and write operations over-run the intended size of the operation, such that writing/reading to/from a buffer over-runs the buffer's boundary and over-writes/over-reads adjacent memory loca-

tion. Consequently, heap over-run and stack over-run fall under the same type of vulnerability. It can be argued that this involves the memory as the shared resource between different pieces of software (e.g. software under the same process). These pieces of software may not necessary communicate between one another or have common functionalities. In matter of fact, studies of buffer over-flow vulnerabilities show [101] that the victims of buffer over-flow vulnerabilities are in many occasions software modules that are not related to the targeted software module of the exploit, but what happen to be residing in adjacent memory area. As a result, when privilege separation is present it should prevent an exploit in a software module from obtaining data of another. In addition, a software module should be able to identify unauthorised memory write to part of its region. Those two properties should be provided by TEE technology (e.g. Intel SGX).

5.4.1.2 Gain Information

This relates to the ability to gain information either directly through call of functions that lead to confidential information or indirectly through reading the memory of software module, changing the address of execution to a desired function [2], or through unauthorised access to confidential data (e.g. memory, cache) by unauthorised software due to an exploit. For instance, buffer over-read can lead to information leakage. Many exploits of this type of vulnerability rely on the absence of sufficient memory protection to accessing data or code that may lead to sensitive data leakage. Privilege separation using TEE can reduce the impact of such attacks by isolating

sensitive parts and by carefully examining the sensitive data and code inside a TEE. It is important to realise that it is the responsibility of the developer to define the entry function to a TEE. While, it is the responsibility of the TEE to enforce policy on the given input.

5.4.1.3 Code Execution

Code execution refers to the ability of an adversary to get a computer system to a state to execute unauthorised code. Code execution under the same TCB has an attack surface that increases with the size of the TCB. It can be argued that once an adversary is able to get a malicious code to execute on a system (e.g. through code injection), they may be able to get access to sensitive data. In an application context, a malicious code has access to the entire memory space of the same process, thus, can read data (confidentiality) or change code (integrity). A TEE should limit the impact of these types of attack and prevent access to sensitive partitions of the same process.

5.4.2 Partitioning the TCB in applications

To mitigate software vulnerabilities, software partitioning is applied, illustrated, and explained with a concrete example of OpenSSL. However, the approach taken here is applicable to all types of applications that protect secret data. In the trusted part we would like to port sensitive functions and data such as hashing functions,

random number generator, certificates, keys and passwords. The untrusted code will be located out of the TEE with the ability to call protected functions to be executed in TEE. While the untrusted code may be able to request for encryption and decryption services from the trusted code, it is unable to read/write the keys and the cryptographic functions that reside within a TEE to provide these services. The untrusted code may merely call the interface TEE functions for execution. The trusted part is considered as a *Black Box* to the untrusted part, thus, protecting the confidentiality and integrity of the code and data.

The application must be partitioned into several parts by identifying the sensitive partitions that require isolation from other parts of the application. The design guideline is to keep a sensitive partition minimal and within feasibility borders to allow formal verification of the code. While the TEE can protect its execution and secrets from external vulnerabilities, it does not protect against badly written code with flaws. Thus, a partition with small code is a corner stone for designing a secure application and has been long advocated for by Saltzer and Schroeder [72]. However, it is important to bear in mind the efficiency of the execution when partitioning the code. A partition scheme that substantially impairs system efficiency will often be unfeasible regardless of its security characteristics.

In order to reduce the impact that may arise from software vulnerabilities in code, multiple enclaves are used to secure the secretes. Each enclave contains one secret such that each key resides in a separate enclave. For example code using a session key lies in one enclave and code using the private key lies in another enclave.

Multiple enclaves are used per account/user/connection, where each enclave contains the secrets generation relevant code and its relevant key, and one enclave for the code that has high frequency of accessing the code after generation.

5.5 Functional Requirements

The following requirements for developing an application to reduce the impact of exploits of software vulnerabilities are derived from the targeted software vulnerabilities explained in the previous section. These are the requirements of our intended method to be used to perform partitioning:

FR-1 When developing an application, the used method for partitioning must be able to identify breaking line between the different software modules. Such that for two modules A and B, module A may have more sensitive function than module B.

FR-2 The method must be able to point to items allocated in memory.

FR-3 The method must be able to identify a function that directly or indirectly manipulates sensitive information.

FR-4 The method must be able to identify sensitive information.

FR-5 The method must be able to rank the sensitivity and privilege level of software modules.

FR-6 The method must be able to identify objects that might reside in adjacent memory addresses.

5.6 Security Requirements

The following requirements are derived from the functional requirements and the types of vulnerabilities described in section 5.4.

SR-1 The integrity and authenticity of all data and code of a contained software module must be protected from external software modules.

SR-2 The confidentiality of the data of an enclaved software module must be protected from other software modules.

SR-3 The confidentiality of enclaved objects must be protected from read and write directly or indirectly by operations in other enclaves.

5.7 Security Evaluation

This section describes the aforementioned approach in relation to mitigating the types of software vulnerabilities in Section 5.4. It can be argued that using a logical partition scheme, the impact of exploiting a software vulnerability can be reduced. In this chapter, enclaves are used in a logical way to address the impact of software

vulnerabilities. The next chapter will expand on the framework used to apply application TCB partitioning. It must be noted that the aim of this approach is not to ensure availability of security-sensitive code or data, but rather, provide protection to security-sensitive data from exploits of undetected software vulnerabilities. The following subsections describe mitigation of the three software vulnerabilities types from world-class applications.

5.7.1 OpenSSL

5.7.1.1 Mitigating Buffer Over-flow/Gain Information

We use a *MiniServer and the OpenSSL library 1.0.2-beta1* and examine buffer over-read vulnerability in that version of OpenSSL, the *Heartbleed* vulnerability [182]. The MiniServer is a web server that serves multiple clients and provides authentication, and secure communication channel. The MiniServer runs on Linux and uses merely minimal code to establish secure connections with clients. Furthermore, it uses the OpenSSL library for establishing secure connection between the server and the client [115].

Figure 5.1 describes how the Heartbleed vulnerability works. To illustrate this further, a client sends a heartbeat packet to the server to check that the session is still alive, the server stores the packet and sends it over to the client for confirmation. Once the heartbeat packet is sent, the client can request from the server to send that packet to verify the session is still alive. As it can be seen from the figure, the bug

is manifested when a malicious client asks the server for more data than what was originally sent.

In this specific case, a straightforward solution would be to fix the vulnerability when found, whereas the proposed method in this chapter of isolating software partitions from each other aims to counter the over-read class of attacks when a vulnerability is overlooked during the verification process.

There are two main components in the SSL protocol: the handshake protocol and the data exchange. During the handshake, the client and the server generate keys which are unique for each connection session. The session defines a set of cryptographic security parameters which can be shared among multiple connections. For the most part, the handshake protocol allows the server and the client to authenticate each other and to negotiate a cryptographic suit. The handshake protocol consists of several messages exchanged between a client and a server prior to establishing a secure channel. Once this is achieved, it is followed by the data exchanged between client and server.

The software partitions of OpenSSL into enclaves consist of one enclave for private key, session key and relevant cryptographic functions for every user to protect the session. Consequently, all cryptographic functions that can be negotiated during the handshake protocol e.g. RSA, Diffie-Hellman, AES, DES, SHA1, SHA256, are contained inside an enclave.

This approach protects the main parts used for secure communication between the

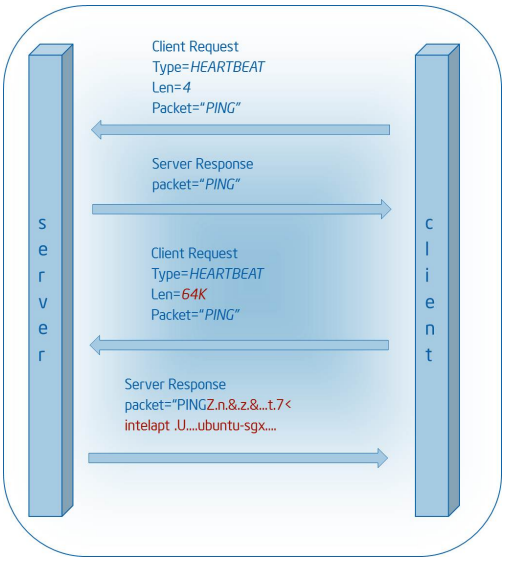

Figure 5.1: Heartbleed Example: An illustration of how the Heartbleed vulnerability can result in buffer overflow

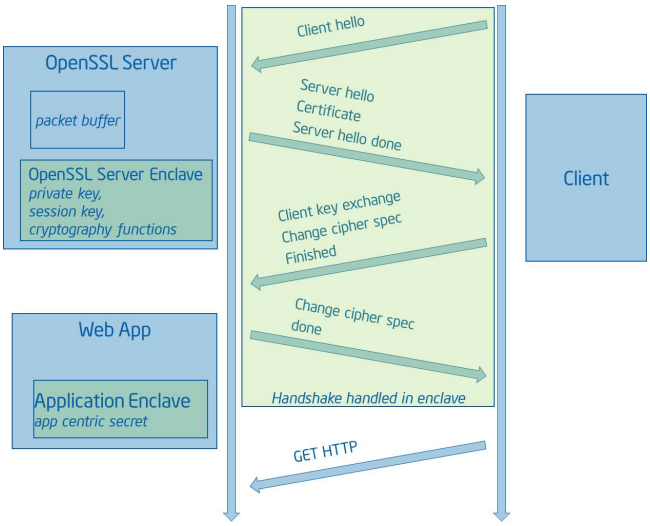

Figure 5.2: OpenSSL Server Enclavised: The design of the software upon enclaving the handshake protocol

client and the server: the private and session keys. However, it does not protect other applications from the Heartbleed vulnerability. Any data residing in the requested memory space may be obtained, thus, it is the responsibility of the application to protect its secrets when secrets reside in the memory.

Figure 5.2 presents the trusted partition of one main element, the handshake protocol, for establishing a secure connection between the client and the server. In the figure, the server application initialises the SSL connections cryptographic suit within an enclave and accepts connections from multiple clients outside the enclave. Whereas the hand shake protocol, private key, session key, and cryptographic functions reside within the same enclave. In this case, I/O operations, e.g. sockets, are executed in normal mode and outside the enclave, thus, no overhead is encountered from switching between enclave mode and normal mode to handle the I/O.

After applying partitioning using Intel SGX enclaves, the Heartbleed exploit could not obtain the session key or the private key. It was merely able to get encrypted data of one of the enclaves. Figure 5.3 shows a screen shot of the tool and the *encrypted data* obtained by the Heartbleed vulnerability. To test the approach chosen to mitigate the vulnerability, we ran a client application using OpenSSL and server application as described earlier. The tool shows the memory region used by OpenSSL on the server side and the packet sent and received. W1 in the figure represents the memory of the OpenSSL during run time. W2 represents the packet sent and received and W3 represents the search results for keys in the packet sent by the server. It can be seen that the tool was successful in finding a key when we attempted to exploit the Heartbleed bug. In a later run after enclavising the Heartbeat code, the tool was not able to infer any keys since keys used by the enclave, the private key and session key, were encrypted in memory.

5.7.1.2 Protecting the Session Key

A consequence of the things discussed in the last section would be that a vulnerability can lead to an exposure of sensitive information in monolithic OpenSSL. To that end, an exposure of the session key can allow an attacker to place himself in the middle and eavesdrop on the communication, enabling him to decrypt sensitive data transmitted during the session. The man-in-the-middle attack as described in [38] allows any unprivileged code to obtain the session key from memory. The attacker exploits the server's unprivliged partition by injecting code. It follows that by intercepting all key

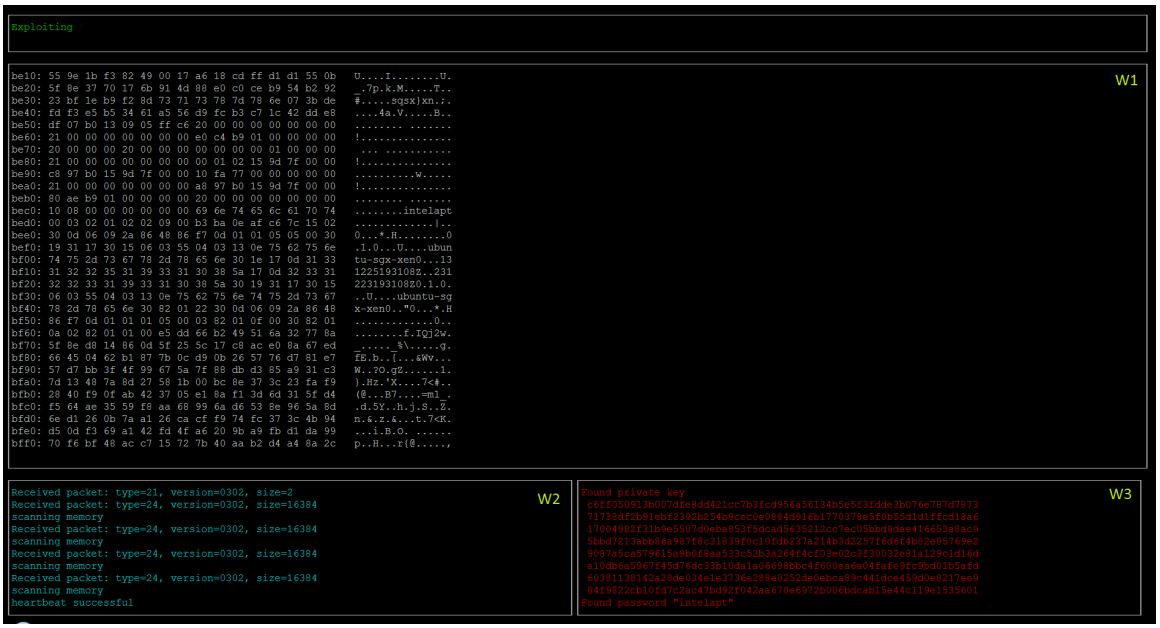

Figure 5.3: Heartbleed Example: Sent and received packets demonstrating how the Heartbleed vulnerability gains access to additional information than originally sent.

exchange messages between the client and the server, an adversary is able to perform an attack, either by intercepting the session key or injecting session key of his choice by placing himself as man-in-the-middle. The attacker then uses the intercepted/injected session key to decrypt sensitive data that might include passwords and IDs. Thus, it is essential to protect the session key and limit access to it. By isolating the handshake protocol in a different TEE, the generation of the key is limited to the handshake code, thereupon external software cannot learn the generated key or any data used during generation. To allow and limit access to the session key, other partitions have to access the handshake partition. To that end, a local attestation between the requestor and requestee is essential prior to allowing a privileged operation within the compartment. In addition, the TEE must not allow operations that may leak information about the secret such as intermediate calculations used to calculate the

key. Once the connection is established, any data coming in or out of the compartment should be encrypted towards the user and towards other partitions in the server. The attacker must not be able in anyway to affect the compartment to send data in the clear once the session key is established. The session between the handshake TEE and other TEEs must be established with local attestation to prevent providing access to a malicious partition (for example a partition that has gone successful code injection exploit). The local attestation using Intel SGX will attest that a partition running inside an enclave and can report on the integrity of the measurement of that specific TEE. Thus, making it a suitable candidate as a mean of authentication between two compartments. In a successful attestation by the handshake compartment, requests to privileged operation are allowed. The local attestation of the handshake compartment to another compartment is essential to prevent impersonating attacks such as an attacker exploit to subvert the authentication mechanism in place.

5.7.2 Gain Information - Android

The Android clipboard is globally accessed by all running apps on the device without any permission requirement. In [183], attacks on Android clipboard are presented. The target of those attacks is clipboard **data manipulation** and **data stealing** attacks which may lead to code injection and **exposure of sensitive data**.

In order to mitigate clipboard attacks as described in Section 2.1, the KNOX clipboard is separated from the user clipboard and each is stored unencrypted under

`/data/clipboard/knox` and `/data/clipboard` respectively. The KNOX clipboard has access to both the KNOX clipboard and Android clipboard. Every application running under KNOX has access to the same KNOX clipboard, thus, there is absolutely no hardware isolation between containers. In other words, if app1 and app2 are running under KNOX simultaneously, both have access to the same clipboard. In addition, the stored clipboard data is not encrypted, leaving the clipboard data susceptible to system exploits. A group of researchers found a vulnerability, *CVE-2016-3996*, that allows disclosure of clipboard data of KNOX containers. The vulnerability is classified as a *Bypassing something* vulnerability which leads eventually to information leakage. The mentioned vulnerability operates by bypassing the protection through calling the direct method to access `/data/clipboard /knox` which does not have a check on the process accessing KNOX clipboard space. We argue that this vulnerability is mainly *Gain Information* vulnerability and the exploit can be easily mitigated by KNOX, have the design been utilising TEE. Namely, applications' data is encrypted in memory when running inside an enclave. Similar approach can be adopted to the clipboard, and to clarify this we describe how using TEE by SGX can mitigate it. The solution can be applied by encrypting KNOX clipboard data, either by having one TEE to run the enterprise workspace, or by having multiple TEEs, one per container. In either case, the KNOX clipboard data is encrypted and the same *bypassing* exploit running from the normal space can get only encrypted data, thus, confidentiality is maintained.

5.7.3 Bypass - Samsung KNOX

Samsung introduced the use of warranty bit in their KNOX supported devices as an indicator of whether a device has been compromised or not. The TrustZone-based Integrity Measurement Architecture (TIMA) ensures this bit is set before instantiating a KNOX container or access to any container in the device. In one example, attackers were able to bypass the main function accessing the TIMA in the *Secure World*, which prevented any access to the KNOX space on rooted devices. Using the Xposed [184] framework to inject code to the *system_server*, attackers managed to override the key function to constantly return the same key, and as a result avoided communication with the TIMA key [55].

The use of KNOX warranty bit depends completely on establishing trust during the boot process, which is a multistep lengthy process that involves verifying several components of the system. It is enough to have one vulnerability in one part of the boot process to compromise the entire system, as it is based on one bit (single point of failure). The aforementioned method did not leverage TEE usage in a good manner, because as mentioned, attackers were able to bypass the main function used to validate this bit, which put the whole application at risk. The advantage of using one TEE for every container, is that each TEE has its own space. For instance, every container's data is encrypted in memory with a different key. Additionally, exploiting an attack inside one container such as information leakage is limited to its container and does not affect others. Leveraging the TEE usage enables a trusted

check of the warranty bit, such as KNOX warranty bit, on the entry point to the container. For instance, when the processor mode of operation is switched to enclave mode in Intel SGX, privileged code (e.g. driver code) executes before the resident code of the application. Similarly, a check of the integrity of KNOX warranty bit in the entire flow of execution by the mentioned privileged code would prevent attackers from exploiting interface function to the TEE as it happened with KNOX.

5.8 Summary

In order to protect the execution of sensitive code and data, it is desirable to use a Trusted Execution Environment (TEE) that is able to report on its code and that does not include untrusted entities such as the OS. To achieve better trustworthiness in a partition, the TCB must be kept small to a level that allows employing measures to guarantee its trustworthiness. Fine-grained TCB partitioning of an application code provides a good means of isolating sensitive parts of the application and defining different trust and privileges between the partitions. Such an approach can protect the execution of a sensitive code from untrusted partitions when access is enforced properly. Intel SGX proves to be a good candidate that keeps the OS out of the TCB and protects the execution of a partition from untrusted code using hardware. It is widely expected that the adoption of technologies like SGX will facilitate the design of secure applications and add another level of protection against various vulnerabilities in the code. In this chapter we have demonstrated *how* such technology can be used

to achieve this. Here, we have explored how to mitigate software vulnerabilities using SGX and will further expand on it in the following chapter.

Another key point is that although the TEE is an important and desirable security feature, it is not a silver bullet against vulnerabilities in code. We demonstrated a logical use of TEE for OpenSSL and discussed potential solutions that can take advantage of Intel SGX to mitigate *bypass* and *gain information* vulnerabilities. However, it can fail in providing such functionality has it been used in a wrong way.

In the next chapter, a broader research investigations is conducted on fine-grained software partitioning using Intel SGX with different applications that includes benchmarking of different schemes. In addition, next chapter presents our proposed framework and performance evaluation considering the points discussed in section 3.1.

The mantra of any good security engineer is: 'Security is not a product, but a process.' It's more than designing strong cryptography into a system; it's designing the entire system such that all security measures, including cryptography, work together

Bruce Schneier

6

TCB Partitioning Framework Design and Implementation

This chapter is based on the following publications:

- A. **Atamli** Reineh, A. Paverd, G. Petracca, and A. Martin, "A framework for application partitioning using trusted execution environments," *Concurrency and Computation: Practice and Experience*, pp. 1–23, 2017

Contents

This chapter builds on the previous chapters 3 and 5. It defines the framework for application partitioning following our investigation with several use-cases, and expands on the three considerations discussed in chapter 3: (1) mitigation of software vulnerabilities in applications, (2) protection from the Operating System (OS), and (3) native performance. First, a framework is proposed and evaluated by classifying previous work according to the proposed framework. Second, we evaluate the framework using three use-cases.

6.1 Application Partitioning Framework

In the previous chapter, a partitioning scheme is defined in order to mitigate the Heartbleed vulnerability. The term *partitioning scheme* refers to a specific design for dividing a particular application into two or more components: (*partitions*). Although these partitions interact with each other, they do so using well-defined interfaces, thus allowing the partitions themselves to be isolated from each other. A *partitioning scheme* is therefore application-specific. For example, a partitioning scheme for the OpenSSL library could be a design document that specifies which OpenSSL functions are moved to the Trusted Execution Environment (TEE). For any given application there are multiple possible partitioning schemes, that differ in terms of which parts of the application are included in TEEs. As a result, these schemes also differ in terms of the security guarantees they provide and their performance impact on the application. At present, application partitioning is usually done on an *ad-hoc* basis,

and is application specific. *Chapter 3* analysed previous work done in the the field and drew conclusions that directly feed into this chapter.

In order to systematise this process, we propose an application-independent framework for application partitioning. The core of our framework is a set of four application-independent *types* into which all application partitioning schemes can be categorised. Instead of proposing individual partitioning schemes, which would be limited to specific applications, the types we define in this framework are based on common characteristics, and can thus be applied across multiple applications. These types are presented in Section 6.1.1. This categorisation is *complete* in the sense that every possible partitioning scheme can be categorised into a type, and it is *unambiguous* in the sense that no scheme can be categorised into more than one type.

In addition to systematising the partitioning schemes, the value of the proposed framework is that we can establish application-independent relationships between the different types of partitioning schemes. These relationships can be used to guide the architectural design of the partitioned applications. These relationships are explained in depth in Section 6.1.2.

6.1.1 Types of Partitioning Schemes

The purpose of partitioning is to enhance the security of an application running on an untrusted platform. The proposed framework therefore focuses on *security-sensitive code* and *security-sensitive data* within the application. Using these concepts

of security-sensitive code and data, we define four types of partitioning schemes.

6.1.1.1 Type 1 - Whole Application

In a type 1 scheme, the majority of the application is included inside a single TEE. This TEE therefore contains application code and data, including all security-sensitive data (e.g., private keys, storage keys, session keys, passwords), as well as all security-sensitive code. The only parts of the application that are not included in the TEE are those that interact with the OS or platform hardware (e.g., manipulating file systems, establishing network connections etc.), since these operations generally cannot be performed from within the TEE. The use of a single TEE allows this type of scheme to be implemented using technologies that only support a single TEE such as ARM TrustZone. However, an outside observer can learn a lot about the use from hte access pattern, yet this is topic is outside the scope of this thesis.

6.1.1.2 Type 2 - Single TEE

In a type 2 scheme, the application uses a single TEE that contains all security-sensitive data and code. In comparison to type 1 schemes, a type 2 scheme aims to minimise the amount of code in the TEE by only including the security-sensitive code and data. This again will include any code that directly accesses the security-sensitive data. For example, if the application uses a cryptographic key for signing data, only this key and the signing function that uses it will be included in the TEE. Similarly

to type 1 schemes, type 2 schemes only require a single TEE per application.

6.1.1.3 Type 3 - Individual TEEs

In a type 3 scheme, individual TEEs are used for each piece of security-sensitive code and data. This means that a single application may require multiple TEEs. Each TEE contains a single piece of security-sensitive code or a single piece of security-sensitive data and the code that uses it. For example, an application that uses multiple cryptographic keys would have one TEE per key. It is often the case that applications aim to isolate different pieces of security-sensitive data within the same application (e.g., session keys for different users), and a type 3 scheme inherently provides this type of isolation. Unlike type 1 and type 2 schemes, a type 3 scheme can only be implemented using technologies that provide multiple TEEs per application (e.g., Software Guard Extension (SGX)). Furthermore, certain type 3 schemes may require secure communication between different TEEs (i.e. a trusted channel). We categorise this subset of schemes as *Type 3 Advanced*,

6.1.1.4 Type 4 - Hybrid

A type 4 scheme refers to a combination of any of the preceding schemes. Multiple TEEs are used to protect security-sensitive code and data, but it is not always the case that each piece of sensitive code or data will have its own TEE. For example, if an application encrypts data using a symmetric key and then encrypts that key using

an asymmetric key, both keys can be included in the same TEE, since compromise of either key would reveal the encrypted data. Similarly, an application with multiple users can have one TEE per user, in which it keeps multiple secrets for each user. On the other hand, if the application applies the same piece of security-sensitive code to multiple users (e.g., an authentication check), this code might be duplicated in multiple TEEs in order to protect its integrity for individual users. As in type 3 schemes, a type 4 scheme requires multiple TEEs per application. Any type 4 scheme that requires inter-TEE secure communication is referred to as *Type 4 Advanced*.

6.1.2 Relationships between Partitioning Schemes

Since the partitioning schemes themselves are application-specific, it is not meaningful to consider quantitative comparisons of partitioning schemes across applications. However, the types of partitioning schemes defined above are application-independent, and can thus be qualitatively compared across applications in various dimensions, as listed bellow.

6.1.2.1 Number of TEEs

Using the notation $num\_tees(T1)$ to denote the number of TEEs required by a type 1 (T1) scheme, the following relationships can be defined.

$$num\_tees(T1) = num\_tees(T2) = 1; num\_tees(T3) \geq 2;$$

$$num\_tees(T3) \geq num\_tees(T4) \geq num\_tees(T2); \quad (6.1)$$

As explained above, both type 1 and type 2 schemes require only a single TEE. A type 3 scheme involves at least two TEEs[1]. By definition, a type 4 scheme requires the same number or fewer TEEs than a type 3 scheme because multiple pieces of security-sensitive code and data can be combined into a single TEE. However, for the scheme to be distinct from type 2, it must make use of more than one TEE.

6.1.2.2 TEE TCB Size

The size of the Trusted Code Base (TCB) within each TEE is an important consideration as it is directly proportional to the attack surface of the TEE. Using the notation $tcb\_size(T1)$ to denote the size of the TCB (e.g., measured in lines of code) in a type 1 TEE, the following relationship can be defined:

$$tcb\_size(T1) \geq tcb\_size(T2) \geq tcb\_size(T4) \geq tcb\_size(T3); \quad (6.2)$$

By definition, a type 2 scheme will have a smaller TCB than a type 1 scheme

[1]Thus in the rare case of a very simple application with only a single piece of security-sensitive code or data, it does not make sense to define type 3 or type 4 schemes since they would be indistinguishable from type 2 schemes.

because the type 1 scheme aims to include as much of the application as possible, while the type 2 scheme minimises this by including security-sensitive code and data only. Similarly, each TEE in a type 3 scheme will have the minimum possible TEE TCB because it only includes a single secret. The TCB size of a type 4 scheme is smaller than that of a type 2 scheme (because of the use of multiple TEEs) but larger than that of a type 3 scheme (because of the possibility for a single TEE to contain multiple pieces of security-sensitive code and data).

6.1.2.3 Duplication of Code

It is very unlikely that type 1 or type 2 schemes would result in duplication of code because of their use of a single TEE. In type 3 and type 4 schemes, the same code may be executed in multiple TEEs in order to isolate pieces of security-sensitive data from each other or to protect the integrity of security-sensitive code that is run for multiple users. Such property is known in security as the none-interference property, and refers to isolation of information flows from each other (each user can be considered as one information flow).

6.1.2.4 Number of TEE Entries

In all hardware-enforced isolation technologies, transferring control into and out of the TEE takes a non-negligible amount of time (although the exact amount of time depends on the specific technology). Therefore the number of times an application

enters (and correspondingly exits) a TEE is an important metric to consider since it has an impact on the performance of the application. Although the actual number of TEE entries is application and use-case dependent, it is possible to establish relationships between the application-independent types in this regard. Using the notation $num\_entries(T1)$ to denote the number of TEE entries in a type 1 TEE, the following relationship can be defined:

$$num\_entries(T3) \geq num\_entries(T4) \geq \{num\_entries(T1), num\_entries(T2)\}; \tag{6.3}$$

By definition, a type 3 scheme would incur the highest number of TEE entries because accessing each piece of security-sensitive code or data requires an individual TEE entry. A type 4 scheme would have fewer entries, but still more than type 1 and type 2 schemes. The difference between type 1 and type 2 schemes in this regard depends on the specific application and use-case. The qualitative relationships between the types of partitioning schemes are summarised in Table 6.1.

Table 6.1: Qualitative relationships between types of partitioning schemes

	Type 1	Type 2	Type 3	Type 4
Number of TEEs	1	1	≥ 2	$\geq$ *T2*, $\leq$ *T3*
TEE TCB size	Largest	Smaller	Smallest	$\leq$ *T2*, $\geq$ *T3*
Duplication of code	No	No	Possibly	Possibly
Number of TEE entries	-	-	$\geq$ *T2*	$\leq$ *T3*

6.2 Related Work Classification

The previous section presents our proposed framework for application partitioning and privilege separation. Now that the framework is defined, this section classifies the previous work presented in Chapters 3, 4, and 5 according to the four partitioning scheme types. Table 6.2 summarises the classification of related work according to our proposed partitioning schemes.

Isolation between different partitions has been achieved in multiple ways. Some systems enforce isolation between the different partitions through privileged software [62, 185, 186, 187, 71, 37]. Other systems use virtualisation [158], partitions as linked libraries [188], processor hardware assistant [153], or trusted computing [62]. We classify these systems according to the granularity of the partitioning, for example a system that isolates an entire application such as Flicker [153] fits into the first partitioning scheme of whole application as one partition. The second level of protection is isolating privileged code/secrets from the rest of the system ,thus, reducing the TCB is achieved in scheme 2 such is the case for Xen [185] and Privman [71]. However, none of these systems provided isolation per user/account/session as proposed by the third and forth partitioning schemes. For example, in Wedge [37] each connection is handled in a different thread to perform isolation between the different sessions and prevent the leakage of the session key to other sessions. The mentioned approach provides high level of security, however, can greatly impair the performance. For instance, during our partitioning work on the OpenSSL library we found that some

code can be shared between the different partitions, this code doesn't use stored data but uses the passed arguments of the calling method. We use this finding to reduce the number of TEEs, thus, reducing the overhead produced of managing the TEEs. In scheme four, the partitions are optimised to reduce the overhead.

Table 6.2: Partition Isolation Methods Classification to the Four Partitioning Schemes.

	First Scheme	Second Scheme	Third Scheme	Fourth Scheme
Thwarting Memory Disclosure[173]	✗	✓	✗	✗
Flicker [153]	✓	✗	✗	✗
TrustVisor [62]	✗	✓	✗	✗
Multilevel Security[189]	✗	✗	✓	✗
Xen[185]	✗	✓	✗	✗
Microkernel[186]	✗	✓	✗	✗
Chrome OS[187]	✗	✗	✗	✓
Partitioning Using SGX[4]	✗	✗	✗	✓
Privman[71]	✗	✓	✗	✗
Wedge[37]	✗	✗	✗	✓
Codejail[188]	✗	✓	✗	✗
Dune[158]	✗	✗	✓	✗

The approach of application partitioning in this thesis differs in the granularity and feasibility of isolating sensitive code. Most approaches rely on software to isolate the execution of sensitive code from the rest of the system. These approaches face significant difficulties when partitioning the code into trusted and untrusted sections. While it is straightforward to isolate an entire application using SGX, it is still feasible for programmers to partition the code into trusted and untrusted sections even when the application is not modular or privilege-separated. Unlike some hardware-based isolation techniques, SGX enables concurrent execution of more than one secure enclave. This allows applications to use various different partitioning schemes to achieve

the required balance between security and performance.

6.3 Case-Studies

We use three case-studies to evaluate our framework: (I) a MiniServer + OpenSSL library, (II) the Apache web server, and (III) SQLite to examine several software partitioning schemes. In the first two use-cases, two points are considered: mitigation of software vulnerability and protection from the OS. In the last use-case; the SQLite, we consider the mitigation of software vulnerabilities, protection from the OS, and native application performance. The MiniServer is a web server that serves multiple clients and provides authentication, and secure communication channel. For our investigation on vulnerabilities' mitigation we use the module of MiniServer + OpenSSL from a previous work [4]. The MiniServer runs on Linux and uses merely minimal code to establish secure connections with clients. Furthermore, it uses the OpenSSL library for establishing a secure connection between the server and the client [115]. Apache [190] is the most used web server software and incorporates different modules such as ModSSL, OpenSSL, ModAuthzUser, and ModAuthBasic to provide secure communication, authorisation, and authentication.

In order to evaluate the security and efficiency of the proposed schemes we investigate the merits of partitioning in each of the case-studies. On the security side we investigate: 1) the ability of a scheme to protect against vulnerabilities in code such as the Heartbleed vulnerability, 2) the number of trusted channels required between

partitions, and 3) the size of the TCB. Our primary reason for considering these evaluation metrics is their impact on the attack surface. For example, the size of the TCB has a direct impact on the number of vulnerabilities in code. Also, an application with multiple TEEs requires trusted channels for communicating between them, thus increasing the complexity of the system and expanding the attack surface as the number of components that require protection increases. On the efficiency side, the number of TEEs is considered, the number of entries to these TEEs, and the size of each one. Moreover, context switching is required when moving in and out of the TEE, introducing an overhead that increases with the number of TEEs and entries to these TEEs. Thus, in the SQLite use-case, performance graphs are presented of the throughput and the execution time. Previous empirical work [4] investigated the ability of Intel SGX to protect against exploitation of the Heartbleed vulnerability in OpenSSL. Based on the findings from that work, here we investigate some other vulnerabilities that are mitigated using our Intel SGX enabled approach. The chosen vulnerabilities are from the same type class as the Heartbleed vulnerability, memory + information leakage class as classified by CVEDetails [125]. We evaluate the security and efficiency of the proposed partitioning scheme types from section 6.1 and present the calculated results in table 6.7.

6.3.1 First Case-Study: *Apache Web Services*

As of the writing of this analysis, Apache holds the biggest share of the web server market. Apache's success is due to its broad range of functionalities, as it supports

concurrency and can therefore serve a big number of clients. The server can be easily configured by editing text files or using one of the many GUIs that are available to manage those. Due to its modularity, many features that are necessary within special application domains can be implemented as add-on modules and plugged into the server. For the most part, the imported add-on modules are third party libraries like OpenSSL, ModSSL, and others. Unfortunately, these third party libraries might bear vulnerabilities, and once exploited may lead to information leakage or compromising the integrity of some parts of the importing application. An example illustrating that is the Heartbleed vulnerability in the OpenSSL that resulted in leakage of usernames and passwords in many applications such as Apache [191].

Apache has three main security modules [192]: (1) authentication, (2) access control, and (3) network communication using Secure Sockets Layer (SSL)/Transport Layer Security (TLS). The authentication modules allow identification of clients, usually through verifying usernames and passwords against a stored database. The Apache access control module restricts access to resources based on client input, such as the presence of a specific header or the IP address or host-name of the client. Third party modules allow you to restrict access to clients that misbehave. The SSL/TLS protocols allow secure communication between the Web server and the client. However, we discuss partitioning more precisely for the OpenSSL separately in the next section.

6.3.1.1 Type 1 Scheme - Whole Application as One Partition

In the first scheme the entire Apache code and data reside in a single TEE. In this partition, even when OpenSSL is used by Apache, we do not consider the OpenSSL library as part of the partition inside the TEE. It follows that the mentioned design choice allows protecting the Apache from memory related vulnerabilities in the OpenSSL library. Sensitive data of the Apache application, OpenSSL enabled, such as usernames and passwords, cannot be extracted or modified by external software and a vulnerability in the OpenSSL library cannot be exploited to gain information. For example, an exploit of the Heartbleed vulnerability cannot obtain the Apache data residing in adjacent memories to the OpenSSL. The use of a TEE encrypts the data in memory with a key not accessible to OpenSSL, hence, in case of buffer over-read exploit due to being in the same process, only encrypted data is read.

6.3.1.2 Type 2 Scheme - Single TEE

In the second scheme we use one TEE to isolate the authentication and authorisation Apache code. We partition the code such that the code of authenticating the user with its data, and the code that restricts access to users is separated from the rest of the modules. Scheme 2 protects against exploitation of external software such as the Heartbleed vulnerability or vulnerabilities residing in other modules CVE-2014-3583, CVE-2010-2227 [125] since they can not access the data and code which are encrypted in memory as part of a TEE. The TCB is smaller than that of scheme 1. However,

there is no separation between the different threads that authenticate the users or enforce policy on access. There is no mutual exclusion between the different security mechanisms or code that handles different users.

6.3.1.3 Type 3 Scheme - Individual TEEs

In the third scheme we use multiple TEEs to isolate the different modules, and the rest of the code used in Apache. On top of that, we duplicated each module according to the number of users/accounts/queries being processed. For example, the authorisation module instantiate a new copy of the module code and data when authorising access to the user. The consideration behind this design decision is to make separation between the accounts during processing and prevent Cross-Site scripting or any data modification of the original copy of the module code and data (Noninterference property). In the same way, modules are isolated with TEEs and TEEs are instantiated according to the same guidelines. Scheme 3, like scheme 2, protects against exploitation of external software such as the Heartbleed vulnerability or a vulnerability residing in other modules CVE-2014-3583, CVE-2010-2227 [125]. In addition, the separation between the different modules and different users/accounts/queries such as restricting access through the access control module, mitigates previously reported vulnerabilities: CVE-2012-3502, CVE-2014-0085, CVE-2014-0226, CVE-2015-1836.

6.3.1.4 Type 4 Scheme - Hybrid

In the fourth scheme we use multiple TEEs to isolate the different modules, and rest of the code used in Apache. However, unlike the type 3 scheme, we reduce the number of TEEs used. For example, the main code of the Apache, referred to as the worker, provides entry point for all accesses to the application and calls for other modules on demand. Such code doesn't store any data and calls for other modules to execute. The design choice in the type 4 scheme is to reduce the number of TEEs used in order to reduce the overhead switching between the environments. We call for this decision due to the characteristics of the code when joining TEEs as one TEE. The design decision is only allowed when joining TEEs does not introduce a threat to the data processed by the TEE. For instance, when the TEE acts as a "pipe" and only processes data and passes it out of the TEE.

The type 4 scheme, like the type 2 scheme, protects against exploitation of external software such as the Heartbleed vulnerability or vulnerabilities residing in other modules CVE-2014-3583, CVE-2010-2227 and like the type 3 scheme provides separation between the different modules and different users/accounts/queries such as restricting access through the access control module mitigates previous vulnerabilities: CVE-2012-3502, CVE-2014-0085, CVE-2014-0226, CVE-2015-1836. Hence providing a complete protection with minimal overhead.

6.3.1.5 Implementation

The Apache case study was the least demanding from application partitioning point of view. As mentioned earlier, the target was to protect sensitive data such as usernames, passwords, and keys. When considering Apache without OpenSSL, the relevant reported vulnerabilities are found in the authentication phase. Initially, we aimed to isolate every flow, however, there was not much gain without breaking the backward compatibility of the authentication library. Nonetheless, the aim remained to mitigate software vulnerabilities. Thus, the isolation involved an entire library, and the aim was to isolate libraries that may be affected from a vulnerability in a library like the authentication. To test the hypothesis, we used partitioning scheme type 2, with isolation of the entire OpenSSL library in one enclave, and the authentication library in another enclave. The latency that was observed for such partitioning scheme ranged between 20%-60%.

6.3.1.6 Evaluation

This section presents all the relevant vulnerabilities that can be mitigated using scheme type 3 and scheme type 4.

Table 6.3 presents some of the Apache vulnerabilities as reported by the CVE Security Vulnerabilities Datasource [125]. The CVE score presents the severity of the vulnerability and the colouring scheme in each ranges from green to red, where green indicates low impact and red indicates high impact. Table 6.4 presents the ability of

Table 6.3: Apache Vulnerability According to CVSS.

	CVSS Score	Confidentiality Impact	Integrity Impact	Availability Impact	Access Complexity	Vulnerability Type
CVE-2015-1836	7.5	Partial	partial	Partial	Low	Obtain Information
CVE-2014-3583	5.0	None	None	Partial	Low	Buffer Overread
CVE-2014-0226	6.8	Partial	partial	Partial	Medium	Obtain Information
CVE-2014-0085	2.1	Partial	None	None	Low	Obtain Information
CVE-2014-3583	4.3	Partial	None	None	Medium	Obtain Information
CVE-2010-2227	6.4	Partial	None	Partial	Low	Obtain Information

Table 6.4: Apache Vulnerability Mitigated with the Four Partitioning Schemes.

	Grade	First Scheme	Second Scheme	Third Scheme	Fourth Scheme
	Buffer Overflow				
CVE-2015-1836[125]	7.5	✗	✗	✓	✓
CVE-2014-3583[125]	5.0	✗	✓	✓	✓
CVE-2014-0226[125]	6.8	✗	✓	✓	✓
CVE-2014-0085[125]	2.1	✗	✓	✓	✓
CVE-2012-3502[125]	4.3	✗	✗	✓	✓
CVE-2010-2227[125]	6.4	✗	✓	✓	✓

each of the partitioning scheme of our solution to mitigate the vulnerabilities from table 6.3.

6.3.2 Second Case-Study: *OpenSSL*

In this section we use the OpenSSL library to examine the proposed software partitioning schemes. Motivated by our previous work chapter 5 and [4], we chose to mitigate vulnerabilities from the memory and information leakage class, the *Heartbleed* vulnerability [182], to evaluate each scheme. The chosen vulnerabilities will demonstrate the ability of each scheme to meet our objective of protecting the private and session keys. While a straightforward solution is to fix the vulnerability when found, our proposed method of isolating software partitions from each other aims to counter the over-read class of attacks when a vulnerability is missed during the verification process that includes code injection in one partition which can affect the whole software stack.

Figure 6.1 represents an attack exploiting the Heartbleed vulnerability between a client and a server.

To emphasise on the vulnerability discussed in Chapter 5, the vulnerability known as Heartbleed results from a missing bounds check in the heartbeat extension which is a 'keep-alive' mechanism between two endpoints to keep the connection alive. The latter was classified as a buffer over-read vulnerability and it allows more data to be read than was initially negotiated between the client and server, thus revealing secrets

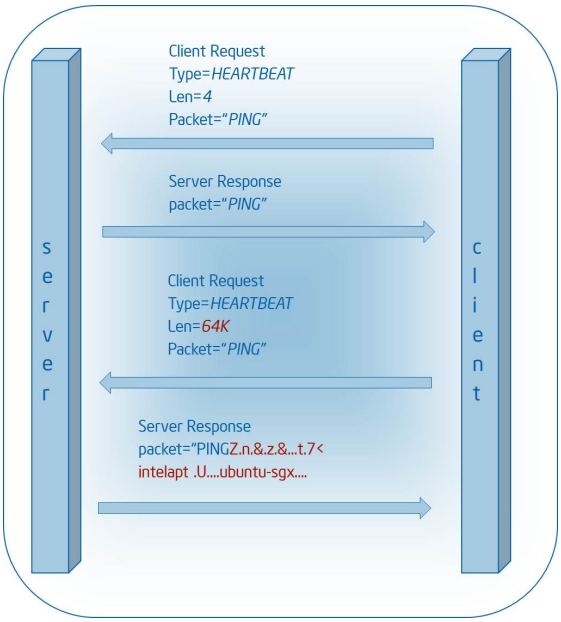

Figure 6.1: HeartBleed Example.

and sensitive data. The sensitive data is not limited to secret keys used within the OpenSSL library, but also includes usernames and passwords of the application that happen to be in the requested memory space. For the most part, applications rely on privileged software such as the OS to prevent external access to an application space. However, in the presence of a vulnerability in an application such as in a third-party library, the OS does not play any part in protecting the data of the entire application, specifically, data that is generated by the application but not used by the imported third-party library.

6.3.2.1 Type 1 Scheme - Whole Application as One Partition

In the first scheme the entire OpenSSL library resides in a single TEE and includes the heartbeat code. The code within a TEE has memory access to every memory address inside the same TEE, thus when a client requests more data than it has

sent, the heartbeat code is still able to extract the requested length, notwithstanding its content e.g., session and private keys, and send it back to the client. Moreover, data from the application using OpenSSL, such as usernames and passwords, can be extracted when residing in adjacent memory addresses to the requested data. Hence, the rest of the application is vulnerable to secrets exposure.

Using a TEE does not protect against vulnerabilities in the code. While the data is protected with encryption from external software when it resides in the memory, it is not protected from vulnerabilities that reside in the TEE. To further illustrate this using the Heartbleed example, the heartbeat code resides within a TEE, thus it is part of the same TCB that contains the secret keys and functions used during the SSL session. As a result, the security properties provided by the TEE are transparent to the contained software, and accessing secrets from an inner function, such as the heartbeat code, can be achieved without the TEE's interference.

Scheme 1 uses one TEE and thus doesn't require any trusted channels. However, the big drawback is the large size of TCB that includes the buffer over-read vulnerability, which in return doesn't protect the confidentiality of secrets upon implementation.

6.3.2.2 Type 2 Scheme - Single TEE

In the second scheme we used one TEE to isolate part of the OpenSSL library including the handshake protocol, private key, and session key. We partition the code

such that only keys handling the code (both session and private) are inside the TEE, but the heartbleed code is outside that TEE.

Scheme 2 protects against exploitation of the Heartbleed vulnerability since the heartbeat code cannot access the session key which is encrypted in memory as part of a TEE. The TCB is smaller than that of scheme 1. However, other secrets of the application, such as the usernames and passwords of the server which are not part of the TEE are not protected. Also, one might question the security of having all the session keys within the same TEE used by the same code. To state the obvious, mutual exclusion between the different sessions is not achieved with this scheme.

6.3.2.3 Type 3 Scheme - Individual TEEs

In scheme 3 each connection has two TEEs, one for the handshake protocol and session key, and one for the data exchange. To elaborate more, since each connection has two TEEs, it's obvious that some duplication of code is inevitable. Nonetheless, the private key resides in a different TEE and can be used by other TEEs that require access to it.

In scheme 3 isolating each secret in a different TEE protects against code vulnerabilities, such as the Heartbleed, that compromise the confidentiality or integrity of the session key or private key. The TCB in each of the TEEs is significantly smaller than in schemes 1 and 2 . However, this approach brings with it other challenges: In order to prevent malicious software from exploiting the different TEEs, a trusted channel

must be established between the different TEEs to assure secure communication and execution of the partitions combined. The latter may impair the execution efficiency in favour of isolating connections. However, more detailed empirical work is needed to examine this, which is beyond the scope of this discussion.

6.3.2.4 Type 4 Scheme - Hybrid

In this approach we considered a hybrid partitioning of the code, which is a combination of the aforementioned schemes. The main code resides in the untrusted space and only a part of the code and data resides in the TEE. The heartbeat code resides in the untrusted space of the application and is thus unable to access the secrets within the TEE. The heartbeat code could reside in a separate TEE if need be. The main focus of our design is on partitioning the application in such a way that sensitive partitions with secrets are isolated from other unrelated partitions. In scheme 4, the TCB is smaller than in schemes 1 and 2 and isolation between the sessions is achieved. However, the TCB is not as small as in scheme 3. The advantage of scheme 4 over scheme 3 is a reduction in the number of TEEs used. The number of trusted channels required between different TEEs is smaller, which results in less overhead in the system and the trusted channel being a target for adversaries. To test this framework, we implemented the hybrid approach using SGX- a combination that proved to be resilient to read over-flow vulnerabilities such as the Heartbleed as presented in chapter 5 and [4]. In addition, with this scheme the size of the TCB inside the enclave proved to be much smaller than scheme 1.

6.3.2.5 Implementation

The OpenSSL case study was very challenging to implement. Splitting the application required hard breaking of the code and implementing some of the the main methods used in the handshake protocol we were trying to isolate. In the original OpenSSL implementation, the handshake protocol is executed using the main process, upon successful handshake the new connection is forked and is executed in a new process. In addition, the parent process maintains a database of the session keys since the handshake protocol executes in the parent process. In order to protect the session key, we had to create a second database inside the enclave. The new design had one enclave that would execute the handshake protocol on new connection. By design, Intel SGX would not enable instantiating the same enclave at the same time, thus, we could not instantiate another enclave for more than one connection without defining it pre-runtime. Thus, for the sake of verifying the success of preventing the leak of the session key, we implemented one connection, which involved extensive engineering efforts. To test the solution, we dumped the memory of the process and looked for sensitive data and also exploited the Heartbleed several times and tried to infer sensitive data from the buffer returned, as shown in chapter 5 Appendix B elaborates more on the implementation details with snap of the code.

6.3.2.6 Evaluation

This section presents all the relevant vulnerabilities that can be mitigated using scheme type 3 and type 4. Table 6.5 presents some of the OpenSSL vulnerabilities as reported by the CVE Security Vulnerabilities Datasource [125]. Table 6.6 presents the ability of each of the partitioning schemes of our solution to mitigate the vulnerabilities from table 6.5.

Table 6.5: OpenSSL Vulnerability according to CVEDetails.

	CVSS Score	Confidentiality Impact	Integrity Impact	Availability Impact	Access Complexity	Vulnerability Type
CVE-2014-0160	5	Partial	None	None	Low	Obtain Information
CVE-2015-3195	5	Partial	None	None	Low	Obtain Information
CVE-2014-3508	4.3	Partial	None	None	Medium	Obtain Information
CVE-2011-0014	5	Partial	None	Partial	Low	Obtain Information

Table 6.6: OpenSSL Vulnerability Mitigation - Four Schemes

	Grade	First Scheme	Second Scheme	Third Scheme	Fourth Scheme
CVE-2014-0160	5	✗	✓	✓	✓
CVE-2015-3195	5	✗	✓	✓	✓
CVE-2014-3508	4.3	✗	✓	✓	✓
CVE-2011-0014	5	✗	✗	✓	✓

In table 6.7 we summarise the analysis of the 4 different partition schemes discussed in the previous section and previous work [4].

Table 6.7: Comparison between the 4 schemes - OpenSSL [4]

	Whole Application	All Secrets	Separate Secret	Hybrid
Number of Enclaves (10 Connections)	1	2	21	11
Trusted Channels between Enclaves (One connection)	0	0	3	2
TCB in enclave	L	S	S	S
Duplication of Code	No	No	Yes	Yes
Capacity Used	M	S	L	M - L

Size Scale: L - Large, M - Medium, S - Small.

6.3.3 Third Case-Study 3: *SQLite*

The last use-case that will be discussed in this chapter is SQLite [193]. A similar approach that relies on client-server partitioning of the code is CryptDB[194], however, unlike our approach, the sensitive part runs on the client device. SQLite is a relational database management system written in the C programming language. SQLite is one of the most used engines in the world and adopted by organisations and cloud providers. The aim of this case study is to show the performance implication when using Intel SGX to isolate sensitive partitions. Initially, we aimed for a partitioning scheme that takes into account the three points discussed in the Venn diagram in

Chapter 3: (1) normal application performance, (2) mitigation of vulnerabilities, and (3) protection from the OS. However, we could not test the mitigation of vulnerabilities due to the lack of relevant reported software vulnerabilities on CVEDetails. Thus, we do not claim optimal solution that addresses the three points in this thesis. However, in this design, the SQL command parser operates mainly on encrypted data flowing from the client session enclave, thus, although historically concerns of vulnerabilities in parsing blocks may be true, it's irrelevant to this design since any data manipulation can be detected by the parsing enclave.

6.3.4 Protecting Users Data

The aim in this work is to protect the confidentiality and integrity of the users' data. Thus, we aim to employ encryption to users' data on the server side opposed to common approaches of employing encryption on the client side [195, 196]. The users' data is encrypted on the server side before writing it to the disk. The design consideration of encrypting the data on the server side is to save the clients from performing the encryption at their end, which requires manageability of the encryption/decryption key used to protect the data. The remote client is merely connected to the server using TLS to assure confidentiality and integrity of the connection. On the server side, the clients' data is received in an encrypted form by trusted and isolated code, which parses the data and encrypts it using the master key. The data is processed in encrypted form in all steps when it's processed outside the isolated regions (the enclave). The data is decrypted only in the isolated region of the enclave and accessed

by the code base of the enclave. For external code, the users' data in memory and on desk is in encrypted form. For this work we adopt the hybrid approach where each client session is handled in different enclaves, with a shared enclave (the parsing enclave) that handles the SQL commands and performs encryption/decryption based on the type of SQL command sent by the client.

Figure 6.2 illustrates the SQLite software architecture and our changes highlighted in green. Our changes in SQLite are mainly in the parser. The received client command is encrypted using the client session enclave with a SQL session key that is generated on each run. The SQL session key is used to encrypt traffic between the client session enclave and the parser enclave. Subsequently, the parser enclave decrypts the SQL command and parses the SQL string. Eventually, it parses the command and encrypts each field using Advanced Encryption Standard (AES)-Galois/Counter Mode (GCM) encryption using the master key. In addition, another column is added to the original table which stores the hash of the values in a row. The parser hashes all fields of each sub-command to allow an integrity check in case a replacement attack occurrs. The reasoning behind the parser enclave being part of the original command parser is for backward compatibility. In the original SQLite, the parser block parses the string command and breaks it down into several sub commands and parses each one based on the column specified on table creation. Once the parser enclave encrypts/decrypts the data, it passes it to the execute function to be written to an SQL file. It is important to realise that the hybrid partitioning scheme allows information flow from several session enclaves to one parser enclave. This design choice limits the

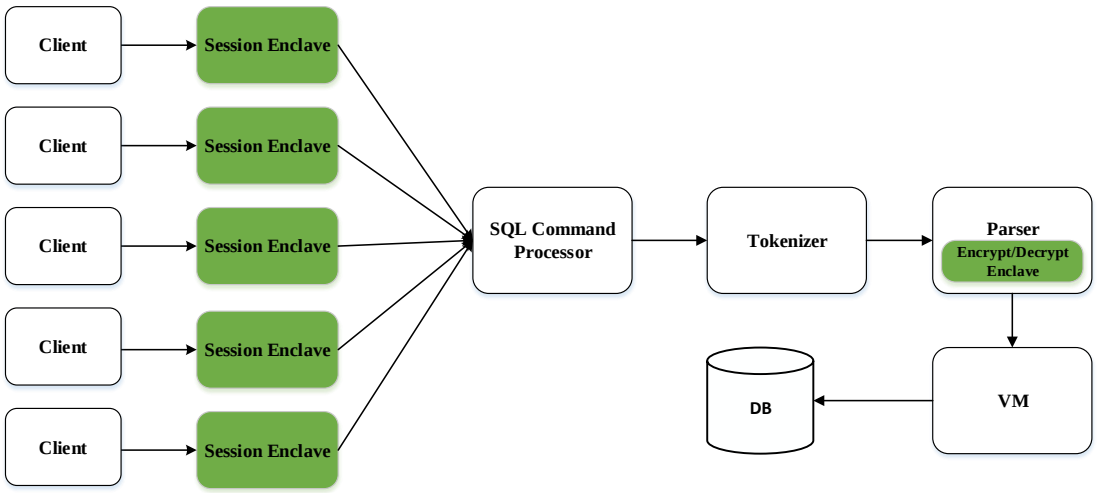

Figure 6.2: Architecture of SecureSQLite

use of the master key for one enclave, hence, smaller TCB.

The approach taken in this chapter is different than other approaches that operate on encrypted data [197, 198, 199, 200, 201, 202, 203]. The aforementioned approaches are limited in the functionality and the aim of this work is to use TEE to limit the code accessing user data while still preventing the service provider from viewing the content being processed by the isolated region (parsing enclave). The code inside the parsing enclave decrypts the data when needed in order to perform the operation. However, it never leaks the data outside the enclave in plain text. It is worth noting that methods that guarantee such properties as mentioned in 3 are outside the scope of this work.

6.3.5 Security and Performance Consideration

In the original implementation of SQLite, an adversary with privileged access to the system is able to view and manipulate the data. Thus, we encrypt data before storing it to the disk. However, even when encrypting each field in the SQL database, an adversary is still able to perform a replacement attack on one of the fields. As a result, a method to assure the integrity of a row was needed, the method allows detection in the event a replacement attack occurs. Initially, encrypting the entire row using AES-GCM would result in Message Authentication Code (MAC) for the entire row which would allow detection of a replacement attack by looking at the MAC for each row. At the same time, such design would require decrypting the entire row for every SELECT operation, which would carry a significant implication on the performance for SELECT operations. For example, consider the command

SELECT *sql* FROM *sqlite_master* WHERE *type*='*table*' AND *Name*='*Paul*'

In order to do the search of all rows with field Name = 'Paul', an entire row of 68 bytes needs to be decrypted to obtain the field "Name", adding an extra step per row compared to the original SQLite. In addition, encrypting the entire row as one block would break the original design of SQLite since the parsing block breaks the SQL command into several sub-commands and values, according to the table setup. Henceforth, each value in a row needs to be encrypted independently of other values and has its own MAC. Coupled with another addition field that was added for each raw, the extra field is a hash of the entire row prior to encryption. For that, Sha256

[204] is used and the parsing enclave checks the integrity of each row only on the final result of the SELECT QUERY. Assuming Sha256 is not broken (such is the case for Sha1 [205]) , is a reasonable assumption for the next few years. However, this might not be the case in ten years time if quantum computing techniques become more of an actual threat [206, 207].

6.3.6 Key Management

There are three types of keys that are used in the described design: client session key, master key, and SQLite session key. The client session key is used for the TLS communication between the server and the client on the other end. Each client has a different key that is generated upon establishing the connection. On the other hand, the master key is used to encrypt the SQLite data prior to writing it to the database on the desk. For the communication between the client session enclave and the parsing enclave, an SQLite session key is generated by the parsing enclave during runtime and is passed to the client session key after preforming dual local attestation. In this way, all traffic between the parsing enclave and the client session enclaves is protected.

Table 6.8: Size of Original Fields and Encrypted Fields

	ID	$Name$	Age	$Address$	$Salary$	$Hash256$
$Example$	1	'Paul'	34	'California'	15000.00	-
$Size before Encryption[bytes]$	8	20	8	20	12	-
$Size after Encryption[bytes]$	36	52	36	52	40	32

6.3.7 Implementation

6.3.7.1 Enclave Cryptography

Each enclave contains cryptographic operations that intend to do one function. On one hand, the client session enclave handles the session between the server and client and encrypts data between the session enclave and parsing enclave. On the other hand, the parsing enclave parses the SQL command and data and updates the database on the disk accordingly. Furthermore, the parsing enclave parses the values of the commands and encrypts them using AES-GCM and adds another column to the table that comprises the accumulative hash of all the values. Each encrypted value has an initialisation Vector (IV), size, MAC and encrypted payload. Table 6.8 describes the sizes of all fields of the values of one row in the table. The encrypted data of each field matches the size of the original field. However, due to the structure of the encrypted data (IV and MAC for each field) the data inflated. Alternatively, the same IV could have been used for the entire database. It can be assumed that storage is relatively cheap, thus, the increase in size is justifiable in favour of increased security.

Finally, SecureSQLite makes use of the cryptographic library provided by Intel SGX with maximum key length of 128 bit AES symmetric key in GCM mode. Each Field value is encrypted separately without dependency on other values with different IV. The outcome of the encryption function is an encrypted blob with MAC for each value which is used to verify the integrity of the data.

6.3.7.2 INSERT Operation

In SecureSQLite design the INSERT operation did not suffer from large overhead. To illustrate this further, the only step added to the INSERT operation is the encryption phase. Following parsing the command, AES-GCM is applied to each of the fields values in the command, and then a hash value is added to the row. In summary, the steps added to the INSERT operation are encryption of each field and hash of the values prior to decryption. The reasoning behind hashing the plain text data prior to encryption is to prevent a replacement attack on a hashed encrypted values. To illustrate this, an adversary with privileged access can view the encrypted database, thus, is able to calculate the hash of the encrypted data. However, an adversary has no way to calculate or deduce the hash of unencrypted values without breaking the cryptography of AES or obtaining the master key.

6.3.7.3 SELECT Operation

The design of SecureSQLite favours SELECT commands by design. To minimise the overhead incurred from the encryption/decryption operations of an entire row, each value is encrypted separately and has its own MAC value. As a result, the size of each field grew larger. For example, the ID field which was originally 8 bytes became 36 bytes due to the IV and MAC. The advantage of such a design choice can be explained when considering the SELECT operation. For example when executing the following SQL command only the field Name is decrypted compared to decrypting the entire row.

SELECT *sql* FROM *sqlite_master* WHERE *type*='*table*' AND *Name*='*Paul*'

The size of the payload needing decryption in such query would be reduced from 5 fields of a payload of 68 bytes to one field of 20 bytes, 3.4 folds of the execution time when decrypting 20 bytes. If the ID was the target in the query, the execution time would be smaller since the encryption code needs to operate on smaller value. Appendix B elaborates further on the programming specifics and structures used in the implementation.

6.3.7.4 DELETE Operation

The DELETE operation implementation is similar to the SELECT operation and would require looking up the rows that match the condition in query. For example,

when considering the following SQL command :

DELETE *sql* FROM *sqlite_master* WHERE *type*='*table*' AND *ID*=*1*

The operation would decrypt only one field, the ID as in the SELECT command. Upon successfully satisfying the query, the row is deleted from the table. To summarise, the only operation that is added to each step is decrypting the requested value of the given field.

6.3.7.5 UPDATE Operation

The UPDATE operation suffered the most in the proposed design. The design choice of encrypting each field value separately caused additional steps of executions to implement the UPDATE operation and made it slightly more complex. Since the additional hash field mentioned earlier is an accumulative hash of all values. For example, when performing the following query:

UPDATE *company* SET *address*='*Texas*' *salary*=*20000.00*

The first step is to find all rows that have "ADDRESS = 'Texas'" (SELECT) which requires decrypting each field separately. In the consequent steps, decrypting an entire row of 68 bytes, updating the new SALARY value and calculating a new hash are performed. Upon finishing, encrypting each row and writing it back to the database. To summarise, one UPDATE operation of one row requires decrypting 68 bytes, calculating new hash using SHA256, and encrypting the new updated value.

6.3.8 Evaluation

In this section we report the results of our experiments. All experiments were performed on a one machine of 8 Cores i7 @3.2GHz, 16 GB RAM, 1 TB. The SGX SDK version is 1.9 and has the latest driver provided by Intel SGX on github [132].

6.3.8.1 SQL Commands

Table 6.9 summarises the result of the execution time of the four operations used to test our implementation in comparison to the execution of the original SQLite without our modification or additional encryption. To calculate the time of execution of a single SQL command, the design was tested using the queries from section 6.3.7. It can be seen from the table that the INSERT and DELETE commands have longer time of execution. It can be argued that the reason for that is slower write to the disk compared to the other operations. The UPDATE operation on one field had less overhead despite the write to disk, this is largely because the write to the desk is a modification of one field and did not require adding an entire row to the table. Further experiments on the UPDATE operation showed longer execution times when there was more than one field to update. For the SELECT operation, the design was tested against a SELECT operation of a unique field: the ID (e.g. WHERE ID == 1). It can be seen from the table that the SELECT operation suffered the second largest overhead among the four operations. This is probably due to the relatively short execution time of SELECT operations, thus, the overhead incurred from decryption

Table 6.9: SecureSQLlite time of execution and overhead

Operation	**SQLite Execution Time (ns)**	**Secure SQLite Execution Time (ns)**	**Secure SQLite Overhead**	**Δ**
INSERT	992	1120	12.23%	12.23%
SELECT	120	174	45.56%	5.64%
DELETE	800	1040	30.34%	9.34%
UPDATE	203	374	84.23%	15.78%

has a more significant impact. Similarly, the UPDATE operation incurred the largest overhead. This can be justified when considering the additional steps mentioned in the implementation section. The additional steps of decrypting the entire row, calculating a new hash, and writing new data to the desk resulted in 84% overhead. Nonetheless, as stated earlier the proposed design favoured SELECT operations and it was built upon minimising the impact on SELECT operations. It can be argued that it is a reasonable assumption for databases that favour read operations such is the case when used for statistics and differential privacy [208, 209, 210].

6.3.8.2 Size of Code Base

In the aforementioned design we described two types of enclaves: the client session enclave and the parsing enclave. The session enclave includes simple TLS that encrypts the traffic between the client and associated enclave on the server side. On top of that the session enclave includes AES-GCM encryption code that encrypts the traffic between the session enclave and parsing enclave. The SQL session key is passed to the session enclave on creation of the session enclave using local attestation. The number of LOC excluding the TLS code is around 2.5k. In the parsing enclave, the

AES code constitute 2k, and with the parsing code of SQLite it is close to 4k LOC. It can be seen that both enclaves have a small code base and can be formally verified.

6.3.8.3 Throughput and Time of Execution

To evaluate the new design, the throughput of the INSERT operation was measured. The evaluation of the throughput aims to test the number of requests per second that can be handled by the server given different payloads. Figure 6.3 presents the throughput of the INSERT operations with different payload sizes. For instance, at 95 bytes the Orginal SQLite has around 79,000 requests per second compared to our modified solution which has 72,000 requests per second. It can be seen from the graph that large payloads result in smaller throughput. In addition, the drop in the throughput in our solution has to do with using the enclave. It is important to note that the implementation frees the memory area allocated in each request upon finishing. This is to say that the size of the enclave can have maximum size of 128MB, thus, for scalability reasons and to prevent crash, the implementation did not use threads, and freed memory upon finishing processing a command. The evaluation included several sizes of payload that ranged from 95 to 4500 bytes, which is equivalent to 1 to 40 sub-commands respectively. Each query included the SQL command, fields and values. Appendix B elaborates further on the SQL commands and structure of the queries.

For the SELECT operation, given our design choice and knowing the implemen-

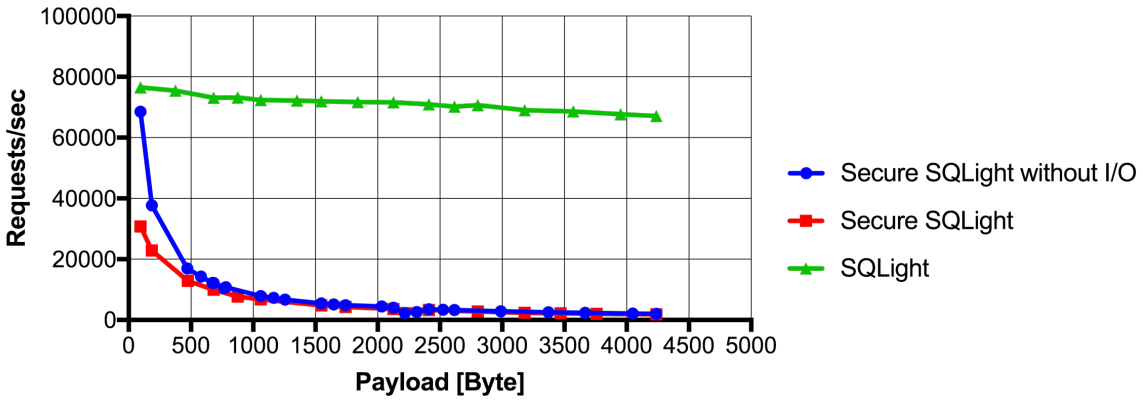

Figure 6.3: Throughput of INSERT Operation in relation to the size of the payload of the SQL command

tation of the command, it was reasonable to calculate the time of the execution given an input of a unique field. For example,

DELETE *sql* FROM *sqlite_master* WHERE *type*='*table*' AND *ID*=*1*

The example above searches for all rows with field ID = 1. Prior to executing the SQL command, the SQL database was initialised with different number of rows for the purpose of the evaluation. Figure 6.4 presents the time of the execution in relation to the number of entries in the SQL database. It can be seen from the graph that the larger the database is, the longer it takes to find a specific entry. In addition, it can be observed from the figure that both original SQLite and SecureSQLite are close to linear representation in some segments, which increases in relation to the size of the SQL database. Notably, there is a sharp glitch around 35 entries. One feasible explanation is that up till 35 entries can be included in one 4k page, and 35 is the edge for transitioning to a new 4k page.

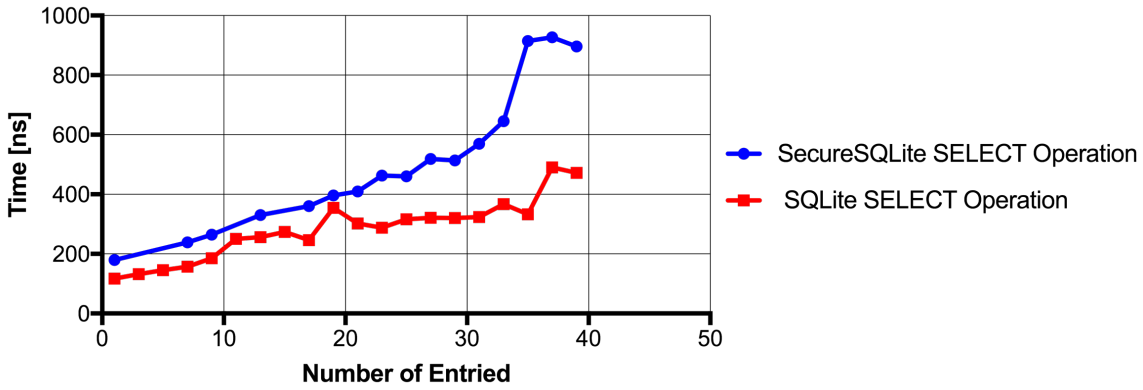

Figure 6.4: Time of Execution of SELECT Operation in relation to the number of entries in the Database

6.3.8.4 Standard Deviation

A noticeable outcome that has been observed from the experiments conducted on SecureSQLite is the variation of the Standard Deviation (SD), particularly when working with the SELECT operation. The experiments showed that the SD increases with the size of the table. Figure 6.5 presents the result of the experiment from the SELECT operation. It can be observed that for SecureSQLite the SD slightly increases in relation to the size of the SQL database. In addition, it can be observed that around the 35 entries there are few glitches in accordance with figure 6.4. Thus, it can be argued again that this has to do with the page size boundary. When an additional page was needed to load the entire database, a glitch is observed. As mentioned before, the enclave memory limitation is 128 MB and for large SQL tables it is not possible to load the entire table in the enclave. As a result, an enclave memory management is needed to load the table in several segments.

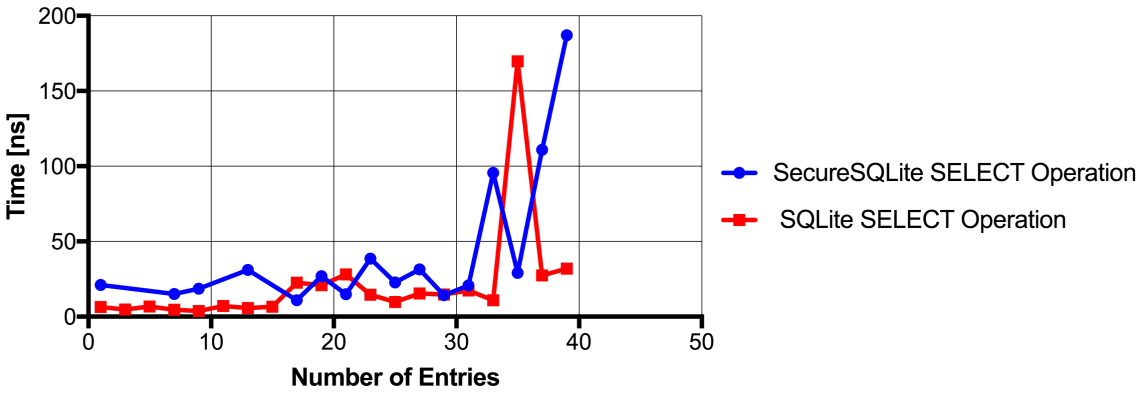

Figure 6.5: Standard Deviation of SELECT Operation in relation to the number of entries in the SQL database Database

6.4 Summary

This chapter has presented a formalised framework for application partitioning concepts and considerations, and has demonstrated how to mitigate software vulnerabilities and secure applications using TEEs. In order to secure applications, it is necessary to consider three aspects: (1) the ability to mitigate software vulnerabilities, (2) protection from the OS, and (3) application native performance. To achieve that, it is best to use TEEs that are rooted in hardware and cannot be tampered with by the OS and external software. For the purpose of formalising the schemes and concepts of application partitioning, a framework has been developed and used to classify all previous work on splitting application and privilege separation. That is to say that the granularity of the partitioning schemes of all related work, does fit the classification to the four partitioning schemes proposed in this chapter. The proposed framework is demonstrated with three case-studies in which Intel SGX has been used to enhance the security of each application. Subsequently, each case-study

was evaluated against the considerations and hypotheses.

In OpenSSL and Apache, the library consisted of many different blocks. As has been seen with the Heartbleed vulnerability, such vulnerability does not only affect the sensitive data of the library, but also the application that is using that library. For instance, in many servers around the world, the exploit was able to obtain OpenSSL session keys, and usernames and passwords used by Apache. The obvious solution is to isolate the application from OpenSSL. However, this solution is not scalable given current technologies, for instance Intel SGX has a limitation of 128MB on the size of the enclave. In addition, it does not mitigate the threat completely. To that end, the partition schemes used in Chapter 5 and 6 demonstrate the ability to mitigate some types of software vulnerabilities and are investigated against all previous exploits of the same class of vulnerabilities as the Heartbleed: *buffer overflow*. Our investigation proved that partitioning application can reduce the impact of all relevant previous vulnerabilities in both use-cases.

In SQLite, the users wish to protect the confidentiality of their data when using cloud service providers, however, how can the clients be sure that the confidentiality and integrity of their data is kept on remote servers? Even when applying cryptography by the server to the users' data, a strong adversary can still manipulate the system, thus, using cryptography does not protect the data when it is being processed. The proposed solution in this thesis, protects the data by adding encryption to the parser block in SQLite and isolating only parts of the code in the parsing block. In addition, for backward compatibility and performance reasons, the isolated

parsing block handles requests from all users. However, a different information flow was created by adding client session enclave to each user to satisfy the property of non-interference between the users. The reported results on the throughput and time of execution of modified SQL commands show a slight overhead due to additional layer of encryption added. Nonetheless, the results confirm the two hypotheses of this thesis and enhance the security of SQLite, and the observed overhead is still justified considering the added protection achieved.

Overall, this chapter therefore confirms the second primary research hypothesis by demonstrating that TEEs in general and Intel SGX in particular can be used to enhance overall application security, not only by protecting the code and data running inside an enclave from the OS and external software, but also by reducing the impact of software vulnerabilities and protecting confidential data from exploits within the same applications. Furthermore, the results from the third case-study confirm that application native performance is only marginally affected when the partitioning scheme is mindful.

We should treat personal electronic data with the same care and respect as weapons-grade plutonium - it is dangerous, long-lasting and once it has leaked there's no getting it back

Cory Doctorow

7

Conclusions

Contents

7.1 Research Context and Hypotheses

Applications growth in functionality has a direct impact on the lines of code in applications. As can be expected, the increase of functionality in applications in recent years brought an increase in the process environment size of code, heap, stack, and loaded modules. As a result of this increase, an increase in the number of vulnerabilities has been observed. Not to mention that most applications run as one process in a monolithic structure under the OS.

In an attempt to address the shortcomings of absence of protection for sensitive code in monolithic applications, the research space of privilege separation became a necessity in security. The area referred to as *privilege separation*, its reach extends more than just applications, and is also used in the realm of OS security. However, the focus on applications is very little although bearing potential benefit. Splitting monolithic applications into several partitions with different privileges can be enforced using different primitives of hardware and software. The approaches differ in the isolation primitives which are mostly done in software. The shortcoming in isolation is believed to be the result of weak isolation measures by the OS [10, 11, 12, 13, 14].

Several pieces of research attempted to address the challenge of splitting applications into several privileged compartments [37, 69, 70, 110, 38, 111, 112] using *default-deny* models in which compartments are given reduced privileges [102, 38], or using *master-slave* models in which the privileges are granted by the master based on authentication and allowed access list. As mentioned, these approaches are software-

based and assume a strong trust model.

As a new approach for enhancing the security of applications, this thesis introduces the use of Trusted Execution Environment (TEE) to mitigate software vulnerabilities and limit the impact of their exploits. There are two benefits of using TEE: first, it guarantees the confidentiality of the data, and second, it can deter attacks on application memory. This enables using TEE for more than just securing the execution of an entire application. It facilitates defining finer-granularity of privileges in an application, and protection by design against possible vulnerabilities. In addition, the TEE grants local and remote attestation rooted in hardware. Thus, an application can attest software partitions prior to passing data for execution. As a result, applications trustworthiness can be attested by the hardware and no software interference and data manipulation can go undetected.

7.1.1 Research Hypotheses

In view of this context, this thesis portrays an investigation of two primary research hypotheses as posed at the beginning of the dissertation:

1. In applications, TEE can be used to enhance overall security and mitigate software vulnerabilities in complex and large scale applications while maintaining the primary functionality and performance requirements of the application.

2. Partitioning patterns can be used to mitigate a multitude of vulnerabilities and

limit the threats of the attack vectors for an application.

The research in this thesis has focused on and attempted both to verify and falsify each of the mentioned hypotheses. As a matter of fact, this thesis determined the extent to which the hypotheses are true by identifying the requirements and examining several use-cases.

This research answered questions such as whether TEEs can be used to mitigate software vulnerabilities, and went beyond that and answered which types of vulnerabilities can be protected. At the same time, this thesis identified the functional and performance limitations of TEEs through evaluation of two hardware technologies: Intel SGX and ARM TrustZone. Likewise, this research identified the current gaps in the approaches to privilege separations. From this perspective, this research identified the main considerations for application partitioning that future solutions should consider. The outcome of this research has confirmed both hypotheses and resulted in four contributions, as stated in the next section.

7.2 Main Contributions

7.2.1 Gaps Analysis of Applications Partitioning

In order to enhance the security of applications, the underneath layers need to provide the necessary primitives to secure them. The most common method used to provide

these primitives relies on software enforced by the OS. The first main contribution in this thesis, presented in *Chapter 3*, is the gap analysis of partitioning applications into several privileges.

By analysing previous approaches to privilege separation, it can be concluded that there is a lack of methodologies and schemes for application partitioning into compartments. As has been noted, it is possible to group the work done into three different groups, it's very hard to extract a pattern to partitioning that provides protection or even prevents an attack by a strong adversary that has full control over the system. In this respect, employing the Citation Tracking method [124] to identify relevant papers in the field did not only give broad view of privilege separation, but also lead to identifying the three considerations which have been overlooked by the research community for a long time. The three considerations when designing a partitioning scheme for application are: (1) mitigation of software vulnerabilities, (2) protection from the OS, and (3) native application performance. Those were the pillars that have guided the performance and security evaluation of the use-cases in subsequent chapters. Without those guidelines, partitioning applications into different privileges would remain an ad-hoc solution that is application specific.

7.2.2 Securing Authentication Protocols

The second main contribution, presented in *Chapter 4*, is a new application design to protect the execution of authentication sensitive code and data in mobile devices. The

authentication protocols presented in the chapter use Trusted Computing (software TPM 2.0) and ARM TrustZone to secure sensitive data and code from tampering. Following identifying and implementing the sensitive parts in the proposed authentication protocol, the time of execution is measured to verify one of the primary hypothesis that the performance of native application is not impaired beyond a level that an application can function. The prototype implementation demonstrates the feasibility of the solution on a mobile platform with the necessary trust characteristics. Unlike, many approaches that assume strong trust to the OS and require significant input from users, our approach is void from those limitation. In addition, the proposed solution showed that it is possible to use TC to secure server-client authentication protocols without requiring many inputs from the users, or multiple security layers such as SMS, or smart cards, which pose a significant limitation that ends up discouraging the users.

In this flow, the use of TPM 2.0 in trusted and untampered execution environment was able to mitigate some of the existing threats on mobile applications. The proposed novel communication protocol between server-client uses both authentication and attestation for setting up the user's account, hence removing the pressure of securing the credential by the user alone. In addition, the work in this chapter points to the deficiencies in using ARM TrustZone technology, and defines the next challenge for the following chapter on mitigation of software vulnerabilities on other platforms.

7.2.3 Mitigating Software Vulnerabilities

The third main contribution, presented in *Chapter 5*, is the first novel design of an application partitioning scheme using Intel SGX, a technology that enables instantiating more than one container. Starting by introducing the issue of vulnerabilities in software, in particular focusing on one example, the Heartbleed vulnerability in the heartbeat code of OpenSSL. The analysis of the buffer over-read vulnerabilities motivated exploring partitioning schemes that bohh (1) reduce the impact of software vulnerabilities and (2) protect sensitive operations from a malicious Operating System (OS). The guidelines adopted by the proposed partitioning scheme were based on the reasonable assumption that *smaller TCB is more feasible to be formally verified against software vulnerabilities*. The study of previous work on privilege separation and software vulnerabilities in *Chapters 3* and *5* respectively, showed that sensitive parts constitute small part of the overall application. Thus, the proposed partitioning scheme demonstrated the ability of protecting sensitive data (e.g. Apache username and password, and OpenSSL session key) from a buffer over-read vulnerability in unrelated software partition, the heartbeat.

Isolating the handshake protocol in OpenSSL, using merely few thousands of LOC, shows the viability of the approach in mitigating some types of software vulnerabilities. In a consecutive step, the proposed approach for mitigating software vulnerabilities is discussed in the context of two types of vulnerabilities found in Samsung KNOX devices, the bypass type, and the gain information type. Without doubt,

our approach portrays the ability to protect these two classes of vulnerabilities when mindful partitioning is done. This proves that the approach can be realised to reduce the impact of software vulnerabilities in many applications.

7.2.4 Application Partitioning Framework and Case-Studies

The fourth main contribution, as presented in *Chapter 6*, is an application partitioning formalisation of the concepts and considerations defined in *Chapter 3*, and three case-studies demonstrating the proposed approach to using TEEs. The formalisation is presented in a framework that provides the underlying structure and tools for reasoning about the main aspects that must be considered when securing application partitions with TEEs: (1) The size of the TCB of a partition, (2) The security characteristics that enable verification of the partition, (3) The different sensitive operations and required privileges of each partition, (4) The partitioning scheme's impact on the application's native performance, and (5) The robustness of the partitioning scheme against privileged software like the OS.

OpenSSL and Apache case-studies demonstrate how our partitioning scheme protects the confidentiality and integrity of the session key from the Heartbleed exploits. Furthermore, the investigation in *Chapter 6* shows the broader potential of partitioning in mitigating vulnerabilities from a similar class of vulnerabilities.

The SQLite case-study shows how employing partitioning to a finer-granularity is not only good for performance, but also essential for good functionality of Intel

SGX. The proposed partitioning scheme allows protection to users' data while making minimal changes to the native application. In addition, the reported performance evaluation shows minor slow down in the execution, which is the result of the encryption code added to protect data-at-rest.

7.3 Future Work

7.3.1 Application Partitioning Tools

The concepts and guidelines defined in this thesis can benefit from tools that ease application partitioning. We analysed information leakage vulnerabilities in three case-studies, a future research should perform broader study on fine-grained software partitioning using Intel SGX with different applications through empirical work, as well as benchmarking each of the scheme types described in this thesis. Furthermore, in order to increase the adoption of software partitioning with isolation enabled, there is a need to ease deploying the partitioning methodologies for developers. The later can be achieved through designing automatic tools with code annotation that perform the partitioning. These tools would rely greatly on methods such as static and dynamic analysis of the code to efficiently deploy the partitioning methodologies. Further investigation which method to adopt and how to use the chosen method in order to build automatic tools that partition the code, will be required.

7.3.2 Partitioning Schemes

In this thesis, the application partitioning framework described different types of partitioning that were used by previous work to separate privileges. The schemes discussed in this thesis, aimed to rebel an attacker that is able to tamper with the execution of the application. However, they still assume trusted OS. It is hard to define an optimal partitioning scheme that would meet all the issues discussed in section 3.4. However, as this thesis demonstrated, different partitioning schemes can satisfy different requirements. Nonetheless, approaches to reduce the impact of vulnerabilities that are different than standard methods of code analysis are very scarce. When the issue of discussion is memory vulnerabilities, TEEs have a lot of potential in reducing the impact of buffer over-flow, memory injection, gain information, and bypass vulnerabilities. Indeed, there is no possible way to point out to every vulnerability in an application. However, it is possible to speculate where vulnerabilities might reside in code. Such knowledge is enough to guide the research on partitioning schemes. Future research should focus on schemes type 3 and type 4, since they are the basis for designing an optimal solution that defends against a myriad of vulnerabilities in application and untrusted OS.

7.3.3 Formal Verification

As has been discussed in this thesis, using TEEs can protect the execution of sensitive code from external software. However, in order to prevent escalation of privileges and

leak of information by one partition, sanity checks of the code must be performed. One of the guidelines in this thesis is to partition an application into multiple compartments to allow for smaller and verifiable TCB.

By defining the characteristics of each container, it is possible to statically verify the security properties of a container. However, without defining suitable partition schemes that make verification feasible, it is not possible to guarantee a reliable method that suits larger set of applications. Moreover, there has been no research on verifying the properties of partitioning schemes and why one partitioning scheme is better than another? Which properties does it satisfy? Future research should address the challenge of verifying a partitioning scheme against the two main considerations presented in this thesis: mitigation of software vulnerabilities, and protection from privileged code like the OS.

7.4 Summary

The objective of this research has been to investigate minimising the Trusted Code Base (TCB) in applications. To achieve the minimisation, a gap analysis of the realm used to achieve that (privilege separation) in applications was conducted in *Chapter 3*, to better understand the considerations when partitioning. The results of chapter 3 gave a clear picture of the shortcoming in the field and the isolation considerations. *Chapter 4* uses the recommendations from chapter 3 and proposes a partitioning scheme that focuses on securing the execution of authentication protocols

in two worlds split environments: trusted and untrusted. To demonstrate the viability of the solution, a performance evaluation is conducted. *Chapter 5* investigates the ability of TEE technology- Intel SGX in mitigating three types of vulnerabilities. To demonstrate the viability of the approach, a partitioning scheme is designed and evaluated against an infamous vulnerability: the Heartbleed. The result confirmed the hypotheses of this thesis. *Chapter 6* builds on chapters 3 and 5 and defines the broader framework for application partitioning. The chapter further confirms the hypotheses through security and performance evaluation of the use-cases.

Chapter 7 discusses various possibilities for future research on privilege separation and how to provide better security for applications using TEEs.

This thesis confirms the primary research hypotheses and therefore demonstrates that TEEs can be used to reduce the impact of software vulnerabilities while instigate an acceptable penalty on the native application's performance.

Bibliography

[1] A. **Atamli** Reineh, A. Paverd, G. Petracca, and A. Martin, "A framework for application partitioning using trusted execution environments," *Concurrency and Computation: Practice and Experience*, pp. 1–23, 2017.

[2] A. **Atamli** Reineh, R. Borgaonkar, R. A. Balisane, G. Petracca, and A. Martin, "Analysis of trusted execution environment usage in samsung knox," in *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, pp. 7:1–7:6, ACM, 2016.

[3] A. **Atamli** Reineh, G. Petracca, J. Uusilehto, and A. Martin, "Enabling secure and usable mobile application: Revealing the nuts and bolts of software tpm in todays mobile devices," *arXiv preprint arXiv:1606.02995*, 2016.

[4] A. **Atamli** Reineh and A. Martin, "Securing application with software partitioning: A case study using sgx," in *Proceedings of 11th International Conference on Security and Privacy in Communication Systems, SecureComm*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pp. 605–621, Springer, 2015.

[5] A. W. **Atamli** and A. Martin, "Threat-Based Security Analysis for the Internet of Things," in *Proceedings of the 2014 International Workshop on Secure Internet of Things*, SIOT '14, (Washington, DC, USA), pp. 35–43, IEEE Computer Society, 2014.

[6] G. Petracca, A. **Atamli** Reineh, Y. Sun, J. Grossklags, and T. Jaeger, "Aware: Preventing abuse of privacy-sensitive sensors via operation bindings," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 379–396, USENIX Association, 2017.

[7] J. R. Nurse, A. **Atamli**, and A. Martin, "Towards a usable framework for modelling security and privacy risks in the smart home," in *International Conference on Human Aspects of Information Security, Privacy, and Trust*, pp. 255–267, Springer, 2016.

[8] G. Petracca, Y. Sun, T. Jaeger, and A. **Atamli**, "Audroid: Preventing attacks on audio channels in mobile devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*, pp. 181–190, ACM, 2015.

[9] A. **Atamli** Reineh, A. J. Paverd, and A. P. Martin, "Trustworthy and secure service-oriented architecture for the internet of things," *arXiv preprint arXiv:1606.01671*, 2016.

[10] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The flask security architecture: System support for diverse security policies," in *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, 1999.

[11] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazieres, "Labels and event processes in the asbestos operating system," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 4, p. 11, 2007.

[12] M. N. Krohn, P. Efstathopoulos, C. Frey, M. F. Kaashoek, E. Kohler, D. Mazieres, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler, "Make least privilege a right (not a privilege).," in *HotOS*, 2005.

[13] K. A. Walsh, *Authorization and trust in software systems.* Cornell University, 2012.

[14] S. V. VanDeBogart, *Modernizing operating system interfaces to improve application security and performance.* University of California, Los Angeles, 2009.

[15] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?," *Computer*, 2006.

[16] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: Security and functionality in a commodity hypervisor," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 189–202, 2011.

[17] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Security and Privacy, 1982 IEEE Symposium on*, pp. 11–11, IEEE, 1982.

[18] W. S. Humphrey, *A discipline for software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[19] P. C. Attie, D. H. Lorenz, A. Portnova, and H. Chockler, "Behavioral compatibility without state explosion: Design and verification of a component-based elevator control system," in *CBSE*, vol. 4063, pp. 33–49, Springer, 2006.

[20] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 37–49, 2007.

[21] D. Bell, L. J. LaPadula, M. Ben-Ari, G. Benson, G. Benson, B. Appelbe, I. Akyildiz, C. Date, D. Denning, P. Denning, *et al.*, "Secure computer system unified exposition and multics interpretation," *Communications of the ACM*, pp. 271–280, 1988.

[22] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, pp. 236–243, 1976.

[23] M. Kuhn, "Openssh pam conversation memory scrubbing weakness, 2003."

[24] A. Ellis, "Heartbleed Update (v3)," [13 April 2014].

[25] L. Deri, "Droplets: Breaking monolithic applications apart," 1995.

[26] S. F. Smith and M. Thober, "Refactoring programs to secure information flows," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, pp. 75–84, 2006.

[27] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[28] J. Viega, J. Bloch, and P. Ch, "Applying aspect-oriented programming to security," in *Cutter IT Journal*, Citeseer, 2001.

[29] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard tm: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, vol. 12, pp. 91–104, 2003.

[30] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pp. 158–168, 2006.

[31] N. Seixas, J. Fonseca, M. Vieira, and H. Madeira, "Looking at web security vulnerabilities from the programming language perspective: A field study," in *2009 20th International Symposium on Software Reliability Engineering*, pp. 129–135, 2009.

[32] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 421–428, IEEE, 2010.

[33] S. Niu, J. Mo, Z. Zhang, and Z. Lv, "Overview of linux vulnerabilities," in *2nd International Conference on Soft Computing in Information Communication Technology*, Atlantis Press, 2014.

[34] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, (New York, NY, USA), pp. 10:1–10:1, ACM, 2013.

[35] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean Application Compartmentalization with SOAAP," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 1016–1031, ACM, 2015.

[36] N. Carlini, A. P. Felt, and D. Wagner, "An evaluation of the google chrome extension security architecture.," in *USENIX Security Symposium*, pp. 97–111, 2012.

[37] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting Applications into Reduced-privilege Compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, (Berkeley, CA, USA), pp. 309–322, USENIX Association, 2008.

[38] P. Marchenko and B. Karp, "Structuring protocol implementations to protect sensitive data.," in *USENIX Security Symposium*, pp. 47–62, 2010.

[39] B. M. List, "Vulnerabilities by bugtraq id," 2004.

[40] M. Howard and D. LeBlanc, *Writing secure code*. Pearson Education, 2003.

[41] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities.," *IEEE Transactions on Software Engineering*, pp. 772 – 787, 2011.

[42] O. Alhazmi, Y. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, pp. 219–228, 2007.

[43] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *ACM SIGSOFT Software Engineering Notes*, pp. 97–106, 2004.

[44] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, pp. 76–79, 2004.

[45] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pp. 421–430, 2007.

[46] U. Steinberg and B. Kauer, "Nova: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

[47] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[48] V. Prokhorenko, K.-K. R. Choo, and H. Ashman, "Web application protection techniques: A taxonomy," *Journal of Network and Computer Applications*, vol. 60, pp. 95–112, 2016.

[49] N. Sullivan, "Staying ahead of OpenSSL vulnerabilities — CloudFlare Blog," 2014.

[50] S. C. Misra and V. C. Bhavsar, *Computational Science and Its Applications ICCSA 2003*, vol. 2667 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, June 2003.

[51] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[52] F. Sabahi, "Cloud computing security threats and responses," in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pp. 245–249, IEEE, 2011.

[53] A. Singh and M. Shrivastava, "Overview of attacks on cloud computing," *International Journal of Engineering and Innovative Technology (IJEIT)*, vol. 1, no. 4, 2012.

[54] G. Pék, L. Buttyán, and B. Bencsáth, "A survey of security issues in hardware virtualization," *ACM Computing Surveys (CSUR)*, p. 40, 2013.

[55] U. Kanonov and A. Wool, "Secure containers in android: the samsung knox case study," *arXiv preprint arXiv:1605.08567*, 2016.

[56] P. Mihailescu, "The fuzzy vault for fingerprints is vulnerable to brute force attack," *arXiv preprint arXiv:0708.2974*, 2007.

[57] A. Adler, "Vulnerabilities in biometric encryption systems," in *International Conference on Audio-and Video-Based Biometric Person Authentication*, pp. 1100–1109, Springer, 2005.

[58] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 38–49, 2012.

[59] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 114–129, 2014.

[60] M. Moixe, "New tricks for defeating ssl in practice," in *BlackHat Conference, USA*, 2009.

[61] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz, "Evolution and growth in large libre software projects," in *Principles of Software Evolution, Eighth International Workshop on*, pp. 165–174, 2005.

[62] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 143–158, May 2010.

[63] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," *SIGPLAN Not.*, 2013.

[64] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," *SIGPLAN Not.*, 2014.

[65] B. Kauer, P. Verissimo, and A. Bessani, "Recursive virtual machines for advanced security mechanisms," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pp. 117–122, 2011.

[66] R. Toegl, M. Pirker, and M. Gissing, "actvsm: A dynamic virtualization platform for enforcement of application integrity.," in *INTRUST*, pp. 326–345, Springer, 2010.

[67] ARM, "ARM TrustZone."

[68] Intel Corporation, "Software Guard Extensions Programming Reference," tech. rep., Intel, 2013.

[69] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in html5 applications," in *Proceedings of the 21st USENIX conference on Security symposium*, pp. 23–23, USENIX Association, 2012.

[70] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, pp. 57–72, 2004.

[71] D. Kilpatrick, "Privman: A Library for Partitioning Applications.," in *USENIX Annual Technical Conference, FREENIX Track*, 2003.

[72] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[73] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica, "Cloud Terminal: Secure Access to Sensitive Applications from Untrusted Systems," in *Proceedings of the*

2012 USENIX Conference on Annual Technical Conference, USENIX ATC'12, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 2012.

[74] T. C. Group.

[75] R. SHIREY, "Rfc 4949–internet security glossary.[sl]: Version, 2007," *Citado na*, p. 32, 2015.

[76] J. Hendricks and L. Van Doorn, "Secure bootstrap is not enough: Shoring up the trusted computing base," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, p. 11, ACM, 2004.

[77] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell, "The inevitability of failure: The flawed assumption of security in modern computing environments," in *Proceedings of the 21st National Information Systems Security Conference*, vol. 10, pp. 303–314, 1998.

[78] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on smart phones: attacks, implications and opportunities," in *Proceedings of the eleventh workshop on mobile computing systems & applications*, pp. 49–54, ACM, 2010.

[79] S. Brand, "Department of defense trusted computer system evaluation criteria," *Orange Book; 5200.28-STD*, 2005.

[80] A. Martin, "The ten-page introduction to trusted computing," 2008.

[81] B. Kauer, "Oslo: Improving the security of trusted computing.," in *USENIX Security Symposium*, pp. 229–237, 2007.

[82] S. Pearson and B. Balacheff, *Trusted computing platforms: TCPA technology in context*. Prentice Hall Professional, 2003.

[83] J. Lyle, *Trustworthy services through attestation*. Oxford University, 2010.

[84] I. Corporation, "Intel 64 and IA-32 Architectures Developer's Manual: Vol. 1 — Santa Clara," 2003.

[85] J. Brown and T. F. Knight Jr, "A minimal trusted computing base for dynamically ensuring secure information flow," *Project Aries TM-015 (November 2001)*, vol. 37, 2001.

[86] J. K. Millen, "Security kernel validation in practice," *Commun. ACM*, vol. 19, pp. 243–250, May 1976.

[87] M. D. Schroeder, "Engineering a security kernel for multics," in *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, (New York, NY, USA), pp. 25–32, ACM, 1975.

[88] K. Walter, S. Schaen, W. Ogden, W. Rounds, D. Shumway, D. Schaeffer, K. Biba, F. Bradshaw, S. Ames, and J. Gilligan, "Structured specification of a security kernel," in *ACM Sigplan Notices*, vol. 10, pp. 285–293, ACM, 1975.

[89] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "The Untapped Potential of Trusted Execution Environments on Mobile Devices," *IEEE Security & Privacy*, vol. 12, pp. 29–37, July 2014.

[90] V. Costan and S. Devadas, "Intel sgx explained.," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[91] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, p. 10, ACM, 2016.

[92] M. Hoekstra, "Intel® sgx for dummies (intel® sgx design objectives)," *retrieved Sep*, vol. 3, p. 4, 2014.

[93] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel software guard extensions: Epid provisioning and attestation services," *ser. Intel Corporation*, 2016.

[94] S. Suneja, C. Isci, V. Bala, E. De Lara, and T. Mummert, "Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, pp. 249–261, ACM, 2014.

[95] J. Wang, A. Stavrou, and A. Ghosh, "Hypercheck: A hardware-assisted integrity monitor," in *Recent Advances in Intrusion Detection*, pp. 158–177, Springer, 2010.

[96] R. Riedmüller, M. M. Seeger, H. Baier, C. Busch, and S. D. Wolthusen, "Constraints on autonomous use of standard gpu components for asynchronous observations and intrusion detection," in *Security and Communication Networks (IWSCN), 2010 2nd International Workshop on*, pp. 1–8, IEEE, 2010.

[97] S. Maity and S. Christopher, "Systems and methods for out-of-band booting of a computer," Dec. 6 2005. US Patent 6,973,587.

[98] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[99] D. Byers, S. Ardi, N. Shahmehri, and C. Duma, "Modeling software vulnerabilities with vulnerability cause graphs," in *Proceedings of the International Conference on Software Maintenance, Philadelphia, PA, USA*, 2006.

[100] Y. Shin and L. Williams, "An initial study on the use of execution complexity metrics as indicators of software vulnerabilities," in *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, pp. 1–7, ACM, 2011.

[101] S. Rehman and K. Mustafa, "Research on software design level security vulnerabilities," *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 6, pp. 1–5, 2009.

[102] J. Newsome, D. Song, J. Newsome, and D. Song, "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *In In Proceedings of the 12th Network and Distributed Systems Security Symposium*, 2005.

[103] J.-E. J. Tevis and J. A. Hamilton, "Methods for the prevention, detection and removal of software security vulnerabilities," in *Proceedings of the 42nd annual Southeast regional conference*, pp. 197–202, ACM, 2004.

[104] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 447–456, ACM, 2010.

[105] J. Fonseca and M. Vieira, "Mapping software faults with web security vulnerabilities," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 257–266, IEEE, 2008.

[106] A. K. Ghosh, T. O'Connor, and G. McGraw, "An automated approach for identifying potential vulnerabilities in software," in *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pp. 104–114, IEEE, 1998.

[107] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 771–781, IEEE Press, 2012.

[108] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE transactions on software engineering*, no. 2, pp. 125–143, 1977.

[109] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "Securekeeper: Confidential zookeeper using intel sgx," in *Proceedings of the 17th International Middleware Conference*, 2016.

[110] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *Security and Privacy (SP), 2016 IEEE Symposium on*, pp. 56–71, IEEE, 2016.

[111] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. R. Pietzuch, and R. Kapitza, "Securekeeper: Confidential zookeeper using intel sgx.," in *Middleware*, p. 14, 2016.

[112] D. Geneiatakis, G. Portokalidis, V. P. Kemerlis, and A. D. Keromytis, "Adaptive defenses for commodity software through virtual application partitioning," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 133–144, ACM, 2012.

[113] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: security and functionality in a commodity hypervisor," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 189–202, ACM, 2011.

[114] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation.," in *USENIX Security Symposium*, 2003.

[115] OpenSSL Software Foundation, "OpenSSL Library Version 1.0.2a," [13 April 2014].

[116] Rembrandt, "OpenSSH S/Key remote information disclosure vulnerability," 2002.

[117] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang, "Protecting sensitive web content from client-side vulnerabilities with cryptons," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 1311–1324, ACM, 2013.

[118] M. Foundation, "Mozilla foundation security advisories."

[119] X. Dong, H. Hu, P. Saxena, and Z. Liang, *A Quantitative Evaluation of Privilege Separation in Web Browser Designs*, pp. 75–93. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[120] N. Carlini, A. P. Felt, and D. Wagner, "An evaluation of the google chrome extension security architecture," in *Proceedings of the 21st USENIX Conference on Security Symposium*, pp. 7–7, 2012.

[121] R. Strackx, P. Agten, N. Avonds, and F. Piessens, "Salus: Kernel support for secure process compartments," *EAI Endorsed Transactions on Security and Safety*, 2015.

[122] A. ZooKeeper, "What is zookeeper," 2014.

[123] U. of Southern California, " Organizing Your Social Sciences Research Paper: Citation Tracking," 22 September 2017.

[124] N. Bakkalbasi, K. Bauer, J. Glover, and L. Wang, "Three options for citation tracking: Google scholar, scopus and web of science," *Biomedical Digital Libraries*, vol. 3, p. 7, Jun 2006.

[125] "CVE security vulnerabilities database," [2016].

[126] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres, "Securing distributed systems with information flow control.," in *NSDI*, vol. 8, pp. 293–308, 2008.

[127] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani, "A design and verification methodology for secure isolated regions," in *ACM SIGPLAN Notices*, pp. 665–681, 2016.

[128] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1169–1184, ACM, 2015.

[129] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, "Automated partitioning of android applications for trusted execution environments," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (New York, NY, USA), pp. 923–934, ACM, 2016.

[130] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks.," in *NDSS*, 2014.

[131] J. Lind, C. Priebe, D. Muthukumaran, D. OKeeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, *et al.*, "Glamdring: Automatic application partitioning for intel sgx," in *2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA*, 2017.

[132] Intel Corporation, "Intel SGX for Linux," 30 June 2017.

[133] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, *et al.*, "Scone: Secure linux containers with intel sgx.," in *OSDI*, pp. 689–703, 2016.

[134] E. Hayashi, O. Riva, K. Strauss, A. J. B. Brush, and S. Schechter, "Goldilocks and the two mobile devices: Going beyond all-or-nothing access to a device's applications," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, 2012.

[135] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," in *Workshop on Mobile Security Technologies (MoST)*, 2012.

[136] I. M. Abbadi and A. Martin, "Trust in the cloud," *Information Security Technical Report*, pp. 108 – 114, 2011.

[137] J. H. Huh, J. Lyle, C. Namiluko, and A. Martin, "Managing application whitelists in trusted distributed systems," *Future Generation Computer Systems*, pp. 211 – 226, 2011.

[138] J. Lyle and A. Martin, "On the feasibility of remote attestation for web services," in *2009 International Conference on Computational Science and Engineering*, pp. 283–288, 2009.

[139] V. Scarlata, C. Rozas, M. Wiseman, D. Grawrock, and C. Vishik, *TPM Virtualization: Building a General Framework*, pp. 43–56. Wiesbaden: Vieweg+Teubner, 2008.

[140] A. Carroll, M. Juarez, J. Polk, and T. Leininger, "Microsoft palladium: A business overview," *Microsoft Content Security Business Unit*, pp. 1–9, 2002.

[141] I. Corp., "Intel Trusted Execution Technology," 2008.

[142] M. Nauman, T. Ali, and A. Rauf, "Using trusted computing for privacy preserving keystroke-based authentication in smartphones," *Telecommunication Systems*, pp. 1–13, 2013.

[143] M. Strasser and P. Sevnic, "A software-based tpm emulator for linux," *Semesterarbeit, ETH Zurich*, 2004.

[144] D. Safford and M. Zohar, "A trusted linux client (tlc)," *Technical Paper, IBM Research*, vol. 1, pp. 1–9, 2005.

[145] T. Tong and D. Evans, "Guardroid: A trusted path for password entry," *Proceedings of Mobile Security Technologies (MoST)*, p. 10, 2013.

[146] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert, "Beyond kernel-level integrity measurement: enabling remote attestation for the android platform," *Trust and Trustworthy Computing*, pp. 1–15, 2010.

[147] A. T. Othman, S. Khan, M. Nauman, and S. Musa, "Towards a high-level trusted computing api for android software stack," in *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, p. 17, ACM, 2013.

[148] M. Nauman and T. Ali, "Token: Trustable keystroke-based authentication for web-based applications on smartphones," in *International Conference on Information Security and Assurance*, pp. 286–297, Springer, 2010.

[149] X. Zhang, O. Aciiçmez, and J.-P. Seifert, "Building efficient integrity measurement and attestation for mobile phone platforms.," in *MobiSec*, pp. 71–82, 2009.

[150] J. H. Huh, H. Kim, J. Lyle, and A. Martin, "Achieving attestation with less effort: an indirect and configurable approach to integrity reporting," in *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pp. 31–36, ACM, 2011.

[151] Microsoft, "TSS.Net."

[152] J. McCune, *Reducing the Trusted Computing Base for Applications on Commodity Systems*. PhD thesis, CMU, Pitssburg, USA, 2008.

[153] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, (New York, NY, USA), pp. 315–328, ACM, 2008.

[154] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth, "Reducing tcb complexity for security-sensitive applications: Three case studies," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, (New York, NY, USA), pp. 161–174, ACM, 2006.

[155] D. G. Murray and S. Hand, "Privilege Separation Made Easy: Trusting Small Libraries Not Big Processes," in *Proceedings of the 1st European Workshop on System Security*, EUROSEC '08, (New York, NY, USA), pp. 40–46, ACM, 2008.

[156] V. Prokhorenko, K.-K. R. Choo, and H. Ashman, "Context-oriented web application protection model," *Applied Mathematics and Computation*, vol. 285, pp. 59–78, 2016.

[157] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the Native Beast of the JVM," in *Proceedings of the 17th ACM Conference on Computer and Com-*

munications Security, CCS '10, (New York, NY, USA), pp. 201–211, ACM, 2010.

[158] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe User-level Access to Privileged CPU Features," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 335–348, USENIX Association, 2012.

[159] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, (Washington, DC, USA), pp. 367–378, IEEE Computer Society, 2015.

[160] Q. Do, B. Martini, and K. K. R. Choo, "Enhancing user privacy on android mobile devices via permissions removal," in *47th Hawaii International Conference on System Sciences (HICSS)*, (Waikoloa, HI, USA), pp. 5070–5079, IEEE Computer Society, 2014.

[161] Q. Do, B. Martini, and K. K. R. Choo, "Enforcing file system permissions on android external storage: Android file system permissions (afp) prototype and owncloud," in *13th International Conference on Trust, Security and Privacy in Computing and Communications*, (Beijing, China), pp. 949–954, IEEE Computer Society, 2014.

[162] G. Petracca, A. **Atamli** Reineh, Y. Sun, J. Grossklags, and T. Jaeger, "Aware: Controlling App Access to I/O Devices on Mobile Platforms," *CoRR*, vol. abs/1604.02171, 2016.

[163] G. Petracca, Y. Sun, A. **Atamli** Reineh, and T. Jaeger, "AuDroid: Preventing Attacks on Audio Channels in Mobile Devices," in *Proceedings of the 2015 Annual Computer Security Applications Conference*, ACSAC'15, 2015.

[164] S. Pokharel, K.-K. R. Choo, and J. Liu, "Mobile cloud security: An adversary model for lightweight browser security," *Computer Standards & Interfaces*, vol. 49, pp. 71–78, 2017.

[165] C. J. D'Orazio, R. Lu, K.-K. R. Choo, and A. V. Vasilakos, "A markov adversary model to detect vulnerable ios devices and vulnerabilities in ios apps," *Applied Mathematics and Computation*, vol. 293, pp. 523–544, 2017.

[166] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han, "A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, (New York, NY, USA), pp. 7:1–7:7, ACM, 2015.

[167] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, pp. 20–37, May 2015.

[168] R. Strackx and F. Piessens, "Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 2–13, ACM, 2012.

[169] Y. Cheng, X. Ding, and R. H. Deng, "DriverGuard: A Fine-Grained Protection on I/O Flows," in *Computer Security ESORICS 2011. 16th European Symposium on Research in Computer Security*, ESORICS'11, pp. 227–244, 2011.

[170] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: A Two-way Sandbox for x86 Native Code," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 409–420, USENIX Association, 2014.

[171] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, (Washington, DC, USA), pp. 79–93, IEEE Computer Society, 2009.

[172] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "PixelVault: Using GPUs for Securing Cryptographic Operations," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 1131–1142, ACM, 2014.

[173] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 1607–1619, ACM, 2015.

[174] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic Hardware/Software Partitioning: A First Approach," in *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, (New York, NY, USA), pp. 250–255, ACM, 2003.

[175] A. Azfar, K.-K. R. Choo, and L. Liu, "An android social app forensics adversary model," in *49th Hawaii International Conference on System Sciences (HICSS)*, (Kauai, HI,USA), pp. 5597–5606, IEEE Computer Society, 2016.

[176] Q. Do, B. Martini, and K.-K. R. Choo, "Exfiltrating data from android devices," *Computers & Security*, vol. 48, pp. 74–91, 2015.

[177] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE Symposium on Security and Privacy*, pp. 640–656, 2015.

[178] C. D'Orazio and K.-K. R. Choo, "An adversary model to evaluate {DRM} protection of video contents on ios devices," *Computers & Security*, vol. 56, pp. 94–110, 2016.

[179] Q. Do, B. Martini, and K.-K. R. Choo, "A forensically sound adversary model for mobile devices," *PLoS ONE*, vol. 10, no. 9, pp. 1–15, 2015.

[180] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, (New York, NY, USA), pp. 11:1–11:1, ACM, 2013.

[181] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 8:1–8:26, 2015.

[182] N. Mehta and Codenomicon, "The Heartbleed Bug," [13 April 2014].

[183] X. Zhang and W. Du, "Attacks on Android clipboard," in *DIMVA*, Springer, 2014.

[184] Xposed, "Xposed module repository.." "http://xposed.info".

[185] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, Oct. 2003.

[186] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," in *In Summer Conference Proceedings 1986*, (Pittsburg,USA), pp. 93–112, USENIX Association, 1986.

[187] A. Barth, C. Jackson, C. Reis, T. Team, *et al.*, "The security architecture of the Chromium browser," 2008.

[188] Y. Wu, S. Sathyanarayan, R. H. C. Yap, and Z. Liang, "Codejail: Application-Transparent isolation of libraries with tight program interactions," in *Computer Security - ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, 2012. Proceedings*, (Berlin, Heidelberg), pp. 859–876, Springer, 2012.

[189] The Department Of Defense, "Multilevel Security in the Department Of Defense: The Basics," [1 March 1995].

[190] Apache Web Server, "The Apache HTTP Server Project," [13 April 2014].

[191] S. F. A. S. Samantha Murphy Kelly, Lorenzo Francheschi-Bicchierai and K. Wagner, "The heartbleed hit list: The passwords you need to change right now," [09 April 2014].

[192] Daniel Lopez Ridruejo, "Apache Overview HOWTO," [10 October 2002].

[193] B. Hipp, "SQLite," [9 May 2000].

[194] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 85–100, ACM, 2011.

[195] A. Kaur and M. Bhardwaj, "Hybrid encryption for cloud database security," *Journal of Engineering Science Technology*, vol. 2, pp. 737–741, 2012.

[196] R. C. Jammalamadaka, S. Mehrotra, and N. Venkatasubramanian, "Pvault: a client server system providing mobile access to personal data," in *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pp. 123–129, ACM, 2005.

[197] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing sql over encrypted data in the database-service-provider model," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, (New York, NY, USA), pp. 216–227, ACM, 2002.

[198] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," in *Proceedings of the VLDB Endowment*, pp. 289–300, VLDB Endowment, 2013.

[199] E. Shi, J. Bethencourt, T. H. Chan, D. Song, and A. Perrig, "Multi-dimensional range query over encrypted data," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pp. 350–364, IEEE, 2007.

[200] Z. Yang, S. Zhong, and R. N. Wright, "Privacy-preserving queries on encrypted data," in *ESORICS*, vol. 4189, pp. 479–495, Springer, 2006.

[201] R. Brinkman, J. Doumen, and W. Jonker, "Using secret sharing for searching in encrypted data," *Secure Data Management*, pp. 319–333, 2004.

[202] C. Dong, G. Russello, and N. Dulay, "Shared and searchable encrypted data for untrusted servers," *Data and Applications Security XXII*, pp. 127–143, 2008.

[203] Z. Liu, J. Li, J. Li, C. Jia, J. Yang, and K. Yuan, "Sql-based fuzzy query mechanism over encrypted database," *International Journal of Data Warehousing and Mining (IJDWM)*, no. 4, pp. 71–87, 2014.

[204] J.-P. Aumasson, O. Dunkelman, S. Indesteege, and B. Preneel, "Cryptanalysis of dynamic sha (2).," in *Selected Areas in Cryptography*, pp. 415–432, Springer, 2009.

[205] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full sha-1.," *IACR Cryptology ePrint Archive*, p. 190, 2017.

[206] M. Farik and S. Ali, "The need for quantum-resistant cryptography in classical computers," in *2016 3rd Asia-Pacific World Congress on Computer Science and Engineering (APWC on CSE)*, pp. 98–105, 2016.

[207] L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, *Report on post-quantum cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016.

[208] C. Dwork, A. Roth, *et al.*, "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014.

[209] C. Dwork, G. N. Rothblum, and S. Vadhan, "Boosting and differential privacy," in *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pp. 51–60, IEEE, 2010.

[210] C. Dwork, "Differential privacy," in *Encyclopedia of Cryptography and Security*, pp. 338–340, Springer, 2011.

A

Intel SGX Application Partitioning - OpenSSL

This appendix contains the implementation of the partitioning scheme to mitigate heartbleed, presented in Chapter 5. Included are the EDL file, snaps of the code from OpenSSL that was changed.

A.1 OpenSSL Software Partitioning

Listing 1.1 presents the code which uses the SSL library and we present the partition of the code for establishing a secure connection with clients on the server side. We start by initialising the SSL server parameters followed by opening connections with clients. Once a connection is formed between a client and a server, a secure SSL channel is formed so that the server reads the client's request and responds with the relevant data.

A.1.1 Enclave Functions

Listing A.1: Enclave.edl

```
enclave {

        // Import the Ocalls for trusted mutex
        from "sgx_tstdc.edl" import *;
        include "types.h"
        include "datatypes.h"

    trusted {
            public int  ecall_ssk_server_init();
            public int  ecall_ssl_accept();
            public int  ecall_ssl_exchage_ctx();
            public int  ecall_ssl_set_fd(ssl* ssl);
    };

    untrusted {
        void print([in, string] const char *string);
    };
};
```

A.1.2 Partitioning Code

Listing A.2: C version

```
// Server Init
ecall_ssl_server_init();
accept();
// Server Hello:  Establishing security capabilities
including initial random number, CipherSuite, Session ID
ecall_ssl_accept();
// handshake Protocol
ecall_ssl_exchange_ctx();
// setting up the input buffer for receiving packets
ecall_ssl_set_fd(ssl);
SSL_setup_buffers(ssl);
// Reading and decrypting the packets
SSL_read(ssl);
// Writing and encrypting the packets
SSL_write(ssl);
SSL_shutdown(ssl); //Closing SSL session
close(client); //Closing connection
```

A.1.3 Handshake Protocol

Listing A.3: s3_pkt.c - SSL_read

```
SSL_read(ssl,ssl_record){
     ...
     SSL_read_bytes(ssl){
          // extract mac_size from the hash
          if (ecall_EVP_MD_CTX_md(ssl->read_hash) != NULL) {
               mac_size = ecall_EVP_MD_CTX_size(ssl->read_hash);}
          ...
          if (ecall_CIPHER_MD_CTX_mode(ssl->enc_read_ctx)
                                        == EVP_CIPH_CBC_MODE) {
          /* We update the length so that the TLS header bytes
          * can be constructed correctly but we need to extract
          * the MAC in constant time from within the record,
          * without leaking the contents of the padding bytes */
           ecall_ss3_cbc_copy_mac(mac,ssl_record,mac_size,orig_len);
          }
          // Extracting the message digest
          ssl->method->ecall_ssl3_dec->mac(ssl,1);
               ...
     }
     ...
}
```

Listing A.4: s3_pkt.c - SSL_write

```
SSL_write(ssl,buffer){
     ...
```

```
    do_ssl3_write(ssl,char * buffer){
        //extract MAC size
        mac_size = ecall_EVP_MD_CTX_size(ssl->write_hash)
        ...
        if (ssl->method->ecall_ssl3_enc->mac(ssl,
        &buf[ssl3_record->length +
        EVP_CIPHER_CTX_iv_length(ssl->enc_write_ctx)],1) < 0)
        {
        goto err;
        }
        //encrypt packet
        ssl->method->ecall_ssl3_enc>enc(ssl,1);
        ...
    }
}
```

B

Encrypted SQLite

B.1 Enclave Functions

Listing B.1: Enclave.edl

```
enclave {

        // Import the Ocalls for trusted mutex
        from "sgx_tstdc.edl" import *;
        include "types.h"
        include "datatypes.h"

    trusted {
    public int  initialize_enclave([in]struct sealed_buf_t* sealed_buf);
    public int  increase_and_seal_data(size_t tid,
                [in, out]struct sealed_buf_t* sealed_buf);
    public int      encrypt_sql_command([user_check] char * command
                ,[user_check] ciphertext_values_t * encrypted_command,
                    size_t cmd_size,uint32_t iter);
    public int  decrypt_sql_command([in,out]char * encrypted_command,
                [in,out] char * command);
    };

    untrusted {
```

```
                void print([in, string] const char *string);
    };
};
```

B.2 SQL Data Structurs

Listing B.2: datatypes.h

```
#include "sgx_report.h"
#include "sgx_eid.h"
#include "sgx_ecp_types.h"
#include "sgx_dh.h"
#include "sgx_tseal.h"

#ifndef DATATYPES_H_
#define DATATYPES_H_

#define DH_KEY_SIZE         20
#define NONCE_SIZE          16
#define MAC_SIZE            16
#define MAC_KEY_SIZE        16
#define PADDING_SIZE        16

#define TAG_SIZE         16
#define IV_SIZE             12

#define DERIVE_MAC_KEY       0x0
#define DERIVE_SESSION_KEY   0x1
#define DERIVE_VK1_KEY       0x3
#define DERIVE_VK2_KEY       0x4

#define CLOSED 0x0
#define IN_PROGRESS 0x1
#define ACTIVE 0x2

#define MESSAGE_EXCHANGE 0x0
#define ENCLAVE_TO_ENCLAVE_CALL 0x1

#define INVALID_ARGUMENT                    -2   ///< Invalid function argument
#define LOGIC_ERROR                         -3   ///< Functional logic error
#define FILE_NOT_FOUND                      -4   ///< File not found

#define SAFE_FREE(ptr)     {if (NULL != (ptr)) {free(ptr); (ptr)=NULL;}}
```

```
#define VMC_ATTRIBUTE_MASK  0xFFFFFFFFFFFFFFCB

typedef uint8_t dh_nonce[NONCE_SIZE];
typedef uint8_t cmac_128[MAC_SIZE];

#pragma pack(push, 1)

//Session Tracker to generate session ids
typedef struct _session_id_tracker_t
{
    uint32_t          session_id;
}session_id_tracker_t;


typedef enum _sgx_sqlite_resp_status
{
    SGX_SQLITE_SUCCESS = 0,
    SGX_SQLITE_MALLOC_ERROR,
    SGX_RAND_FAILED
} sgx_sqlite_resp_status_t;

typedef struct _encrypted_payload
{
    uint8_t iv[12];
    uint8_t mac[SGX_AESGCM_MAC_SIZE];
    uint32_t size;
    uint8_t payload[];
} encrypted_payload_t;

typedef struct ciphertext_values
{
    encrypted_payload_t * ID_ciphertext;
    encrypted_payload_t * Name_ciphertext;
    encrypted_payload_t * Age_ciphertext;
    encrypted_payload_t * Address_ciphertext;
    encrypted_payload_t * Salary_ciphertext;
}ciphertext_values_t;

#pragma pack(pop)

#endif
```

B.3 Encryption/Decryption Stream

Listing B.3: Enclave.cpp

```
int encrypt_stream(uint8_t * payload,uint32_t plaintext_size,
                   encrypted_payload_t * ciphertext) {

    char string_buf[BUFSIZ] = {'\0'};
    uint8_t *payload_data;
    uint8_t ctr[12];
    uint32_t payload_size = plaintext_size;

    //memcpy(&payload_size,plaintext_size,sizeof(uint32_t));

    encrypted_payload_t * enc_payload;

    sgx_key_128bit_t * master_key;
    sgx_aes_gcm_128bit_tag_t mac;


    //Generate the IV
    if(sgx_read_rand(&ctr[0],
        16-4) != SGX_SUCCESS)
    {
        return SGX_RAND_FAILED;
    }
    memset(&ctr[12], 0, 4);
    uint32_t ctr_num_bit_size = 32;


    //Allocate memory for the AES-GCM request message
    enc_payload = (encrypted_payload_t*)malloc(sizeof(encrypted_payload_t)
                    + payload_size);
    if(!enc_payload)
    {
        return SGX_SQLITE_MALLOC_ERROR;
    }

    if (DEBUG_MODE) {
        snprintf(string_buf, BUFSIZ, "\nData2Encrypt >: ");
        print(string_buf);

        for (int i=0;i<payload_size;i++){
            snprintf(string_buf, BUFSIZ, "%x ",payload[i]);
            print(string_buf);
```

```
    }
    snprintf(string_buf, BUFSIZ, "\n");
    print(string_buf);
}
//Generate Provision KEY
//TODO: move to else where
get_provision_key(&secret_key);
master_key = &secret_key;

if (DEBUG_MODE) {
    snprintf(string_buf, BUFSIZ, "\nGenerated 128 Key:>: %x %x\n"
                            ,master_key[8],master_key);
    print(string_buf);
}

memset(enc_payload, 0, sizeof(encrypted_payload_t)+ payload_size);
//mset(enc_payload->payload,0,payload_size);
//initilise the struct

 sgx_status_t  stat = sgx_rijndael128GCM_encrypt(master_key, payload,
                    payload_size,
                    reinterpret_cast<uint8_t *>(&(enc_payload->payload)),
                    reinterpret_cast<uint8_t *>(ctr),
                    12, NULL, 0,
                    &(mac));


if (stat != SGX_SUCCESS)
    return -1;

memcpy(enc_payload->iv,ctr,sizeof(ctr));
memcpy(enc_payload->iv,ctr,sizeof(ctr));
enc_payload->size = payload_size;
memcpy(ciphertext,enc_payload,sizeof(encrypted_payload_t)+ payload_size);


if (DEBUG_MODE) {
    snprintf(string_buf, BUFSIZ, "\nCiphertext >: ");
    print(string_buf);

    for (int i=0;i<payload_size;i++){
        snprintf(string_buf, BUFSIZ, "%x",enc_payload->payload[i]);
        print(string_buf);
    }
    snprintf(string_buf, BUFSIZ, "\n");
```

```
        print(string_buf);
    }
    SAFE_FREE(enc_payload);
    return SGX_SUCCESS;
}

int  decrypt_stream(encrypted_payload_t * ciphertext,uint8_t * plain_text) {

    char string_buf[BUFSIZ] = {'\0'};
    uint8_t *plain_data;
    uint8_t ctr[12];
    uint32_t payload_size = ciphertext->size;

    sgx_key_128bit_t * master_key;
    sgx_aes_gcm_128bit_tag_t mac;


    //Generate the IV
    if(sgx_read_rand(&ctr[0],
        16-4) != SGX_SUCCESS)
    {
        return SGX_RAND_FAILED;
    }
    memset(&ctr[12], 0, 4);
    uint32_t ctr_num_bit_size = 32;

    //Allocate memory for the AES-GCM request message
    plain_data = (uint8_t*) malloc(ciphertext->size);
    if(!plain_data)
    {
        return SGX_SQLITE_MALLOC_ERROR;
    }

    if (DEBUG_MODE) {
        snprintf(string_buf, BUFSIZ, "\nCiphertext >: ");
        print(string_buf);

        for (int i=0;i<ciphertext->size;i++){
            snprintf(string_buf, BUFSIZ, "%x",ciphertext->payload[i]);
            print(string_buf);
        }
        snprintf(string_buf, BUFSIZ, "\n");
        print(string_buf);
    }
```

```
    master_key = &secret_key;


    memset(plain_data, 0, ciphertext->size);


    sgx_status_t stat =  sgx_rijndael128GCM_decrypt(master_key,
                        reinterpret_cast<uint8_t *> (ciphertext->payload),
                    ciphertext->size, reinterpret_cast<uint8_t *>(plain_data),
                    reinterpret_cast<uint8_t *>(ciphertext->iv),
                    12, NULL, 0,
                    &mac);

    if (stat != SGX_SUCCESS)
        return -1;


    memcpy(plain_text,plain_data,ciphertext->size);
    if (DEBUG_MODE) {
        snprintf(string_buf, BUFSIZ, "\nPlain Text >: ");
        print(string_buf);

        for (int i=0;i<payload_size;i++){
            snprintf(string_buf, BUFSIZ, "%x ",plain_data[i]);
            print(string_buf);
        }
        snprintf(string_buf, BUFSIZ, "\n");
        print(string_buf);
    }
    SAFE_FREE(plain_data);
    return 0;
}
```