CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A Framework for Application Partitioning using Trusted Execution Environments

Ahmad Atamli-Reineh[1*], Andrew Paverd[2], Giuseppe Petracca[3], and Andrew Martin[1]

[1]*Department of Computer Science, University of Oxford*
[2]*Department of Computer Science, Aalto University*
[3]*Department of Computer Science and Engineering, Pennsylvania State University*

SUMMARY

The size and complexity of modern applications are the underlying causes of numerous security vulnerabilities. In order to mitigate the risks arising from such vulnerabilities, various techniques have been proposed to isolate the execution of sensitive code from the rest of the application and from other software on the platform (such as the operating system). New technologies, notably Intel's Software Guard Extensions (SGX), are becoming available to enhance the security of partitioned applications. SGX provides a trusted execution environment (TEE), called an *enclave*, that protects the integrity of the code and the confidentiality of the data inside it from other software, including the operating system. However, even with these partitioning techniques, it is not immediately clear exactly *how* they can and should be used to partition applications. How should a particular application be partitioned? How many TEEs should be used? What granularity of partitioning should be applied? To some extent, this is dependent on the capabilities and performance of the partitioning technology in use. However, as partitioning becomes increasingly common, there is a need for systematization in the design of partitioning schemes.
To address this need, we present a novel framework consisting of four overarching *types* of partitioning schemes through which applications can make use of TEEs. These schemes range from coarse-grained partitioning, in which the whole application is included in a single TEE, through to ultra-fine partitioning, in which each piece of security-sensitive code and data is protected in an individual TEE. Although partitioning schemes themselves are application-specific, we establish application-independent relationships between the types we have defined. Since these relationships have an impact on both the security and performance of the partitioning scheme, we envisage that our framework can be used by software architects to guide the design of application partitioning schemes. To demonstrate the applicability of our framework, we have carried out case studies on two widely-used software packages, the Apache web server and the OpenSSL library. In each case study, we provide four high level partitioning schemes - one for each of the types in our framework. We also systematically review the related work on hardware-enforced partitioning by categorising previous research efforts according to our framework. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: SGX; Software Vulnerabilities; Hardware Security; Trusted Execution Environment

1. INTRODUCTION

Over periods of just a few years, applications are seen to grow tremendously in functionality and size. This growth in sensitive applications and libraries, such as the Apache web server and OpenSSL, has 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 these applications, more classes of vulnerabilities have been identified, such as stealing secrets and modifying sensitive

*Correspondence to: E-mail: atamli@cs.ox.ac.uk

code [1, 2]. An example that demonstrates this was the *HeartBleed* bug in the OpenSSL library where an attacker was able to obtain sensitive information including user names and passwords, credentials, and sensitive keys from remote servers [3].

Each discipline has its own type of sensitive data. In the financial sector, software needs to protect credit card details, account numbers, bank account numbers and client financial information. In the personal health sector, the types of sensitive data include insurance information, medical information, social security numbers, addresses and other personal information. In the military and government sectors, software controls may apply to operation information, high profile individuals information, weapons location, and public services access information. In the business sector, sensitive data includes trade secrets, research and business intelligence data, management reports, customer information, and sales data. Despite differing emphasese, all sectors have sensitive data that must be protected.

Much research has considered the design of systems based on well-known operating systems and hardware components to protect sensitive code. Many of these systems leverage virtualisation and trusted computing to isolate the execution of the entire application [4–12]. However, many applications have thousands or millions of lines of code, which makes it hard to gain assurance that no vulnerability exists in the code itself. Moreover, when virtualisation is used to provide isolation between different executions, there are many trust assumptions that make these systems limited in their security properties. For example, the Virtual Machine Monitor (VMM) or the code providing isolation needs to be trusted, loading the Trusted Computing Base (TCB) with thousands of lines of code. The TCB is includes the code that is trusted to enforce the security policy of the system. This code usually runs inside the same environment as sensitive code and data. The isolation of a software partition protects the data and the execution from external code (e.g. the OS) and applications running in the same system. It follows that software partitioning of the application into several trusted and untrusted partitions is expected to produce smaller partitions of code in comparison to the whole application as one partition. When partitioning into smaller chunks is feasible, this improves security because it is easier to audit or even formally verify individual partitions, which are isolated from external code and vulnerabilities, including vulnerabilities in other partitions of the same application.

An isolated environment shares no data unless explicitly designed by the programmer to do so. The aforementioned method of accessing the data through a defined interface prevents unintended privilege sharing. The burden of identifying the data to share lies on the programmer, who also needs to define the method through which the data is shared. An example that demonstrate this is the session handling code of the users in the Apache web services. The session handling code makes use of over 500 memory objects, scattered throughout the process space. Moreover, the usual OS primitives of dividing the execution of the program through forking another process, *exec* to replace the contents of the currently running process, and IPC (Inter Process Communication) between processes to share intended memory objects are by no mean the best method in the aforementioned case. This operation is heavily used by software and suffers from many performance issues.

Other systems [13–20] provide isolation for the execution of a sensitive code block without defining the portion of the application running on the trusted space, the granularity of these approaches to port sensitive code, or the feasibility of porting small code, such as only few methods of an existing library. For instance, the TrustVisor [14] authors appreciate the complexity of porting security sensitive code into a trusted environment. Porting security sensitive code is straightforward if the program is privilege-separated and modular. However, it is a greater challenge in complex applications such as Apache + OpenSSL [14].

To overcome the above mentioned shortcomings, processor extensions have been proposed in several pieces of research [21, 22] to protect software execution and reduce the TCB. Protecting the code execution of the TCB is achieved with a *Trusted Execution Environment* (TEE) in hardware, which prevents external software from tampering with the execution, or modifying an existing code/data. Intel has also proposed security extensions to the Intel Architecture called Intel Software Guard Extensions (Intel SGX) [23, 24], that enable provisioning of sensitive data within applications. These extensions allow an application to instantiate a protected container to ensure

DOI: 10.1002/cpe

the confidentiality and integrity of the data even in the presence of malware, while also relying on hardware to prevent external access to the container's memory area.

The protected container protects the inner code/data from external software, even if privileged, and is referred to as an *enclave*.

Generally, the code and data are freely available for inspection and analysis prior to loading them into the enclave. Once loaded into the enclave and measured, they become protected against external software access. In order to store data outside the enclave's boundary (e.g. on the disk), the application can request the enclave to seal the data beforehand. Furthermore, the platform key, which is used to encrypt the data, ties the data to the platform and can be used to report platform identity to remote parties. Overall, these capabilities extend the ability of enterprises and personnel to design secure applications by relying strongly on hardware instead of traditional software isolation techniques. The aforementioned hardware provides another layer of protection against the exploitation of vulnerabilities missed by the tools verifying the correctness of the code or in manual reviews.

However, even though many such technologies are available, it is not necessarily obvious exactly *how* they can and should be used to protect real applications. In particular, to the best of our knowledge, there is no methodology or set of guidelines that software architects and developers can use to determine *which parts* of an application to encapsulate in an enclave. Modern secure hardware necessarily provides flexibility in this regard. For example, in one of the papers introducing SGX [23], the authors note that:

An application can be encapsulated by a single enclave or can be decomposed into smaller components, such that only security critical components are placed into an enclave.

In some simple cases, the partitioning scheme might indeed be obvious, but as applications increase in size and complexity, the number of possible schemes increases and the choice of the optimal scheme becomes a critical consideration. From a technical perspective, partitioning schemes primarily vary in terms of (1) the security guarantees they provide and (2) their impact on the performance of the application. The choice of partitioning scheme also has other indirect implications, such as the effort of the application developers or software maintainers, but these are beyond the scope of this paper.

In this paper we investigate different partitioning schemes for enhancing the security of applications. Specifically, we consider schemes through which applications can make use of TEEs. As a first step towards systematising application partitioning for TEEs, we define a framework for categorising partitioning schemes into one of four overarching types, based on the number of TEEs used and the elements included in the TEEs. Although the partitioning schemes themselves are application-specific, the *types* we define are application-independent and can thus be applied to all applications. Furthermore, although quantitative measurements of performance and TCB size are application-specific, we establish qualitative relationships between the different types of schemes that are application-independent. We envisage that this framework can be used by software architects and developers when designing partitioning schemes for individual applications. We evaluate our framework using two case studies: the OpenSSL cryptographic library, and the Apache webserver. In each case study, we present a high level overview of four partitioning schemes – one for each of the types we have defined. Using published vulnerabilities for each application, we show which types of partitioning scheme would have prevented specific vulnerabilities.

Although others have presented details of TEEs [13, 14, 23–25], tools for automatically partitioning applications [26], and tools for evaluating compartmentalisation schemes [27], to the best of our knowledge, we are the first to present a systematisation of partitioning schemes.

Our main contributions are:

- 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.

DOI: 10.1002/cpe

- We explain these types of schemes and the relationships between them using two case studies, in which we evaluate the security properties of each type of scheme using published vulnerabilities.
- We categorise previous work on application partitioning schemes using our new framework.

The paper is divided into seven sections. Section 2 provides a brief background on TEEs and software isolation, using Intel SGX as an example. Section 3 describes the security objectives and defines the adversary model. In Section 4 we present our framework and define the four application-independent types of partitioning schemes. Section 5 presents two case studies in which real-world applications are partitioned using the different types of schemes and the security and efficiency of each type of scheme is evaluated. Section 6 discusses related work and shows how all previous work can be categorized using our framework. Section 7 presents our future plans for work on compartmentalisation and Section 8 presents our final conclusions.

2. BACKGROUND

2.1. Secure Software Partitioning

Privilege separation of trusted code from 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 the OS. Although this technique is already in use, it has arguably not yet reached its full potential since new technologies are still being developed to improve it. These include tools that assist with the partitioning of software, as well as technologies for strengthening the isolation between different partitions.

The principle of least privilege [28] could involve 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 about which data manipulated by thier code is sensitive, and a similarly good idea about which code is most risky (typically because it handles user input). So why do so few programmers of networked software divide their code into minimally privileged compartments? As others have noted [2, 6], 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's memory; library calls may leave behind sensitive intermediate results.

2.2. Isolation Mechanisms

In this section we list different mechanisms used for isolation, and provide brief examples of systems that make use of such mechanisms.

2.2.1. Software-Enforced Isolation There are several ways to create separation between partitions. The most common approach is to use privileged code such as an OS or Virtual Machine Monitor (VMM) that enforces access control semantics [14]. 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.

DOI: 10.1002/cpe

Another approach for separation is through performing disaggregation based on dynamic libraries [29]. In this work Murray et al. reduce the Trusted Computing Base (TCB) of the dynamic libraries by excluding the calling process from the TCB. The dynamic library is called on demand to a protection domain that is different from the host process. For this approach to work, it requires the intervention of software such as the OS, kernel, or VMM to achieve isolation between the host process and processes calling the library. The enforcing software (e.g. the OS) would intercept the library calls of processes and perform the appropriate protection domain switch.

2.2.2. Hardware-Enforced Isolation In order to isolate a partition from the rest of the system, various hardware primitives have been proposed to provide a Trusted Execution Environment (TEE) [13, 21]. Unlike a cryptographic co-processor, the TEE is a fully-fledged execution environment that can execute arbitrary code. The TEE isolates the execution of this code from the rest of the system using hardware capabilities of the platform. The TEE also protects the data stored within the TEE from being accessed from outside the TEE. We refer to the code in the TEE as trusted code, and the code of the rest of the system as untrusted code.

One example of a TEE is ARM TrustZone technology [21], which is a set of extensions to the CPU and memory controller that allow the platform to provide a TEE. In TrustZone, the platform provides a single TEE, which is referred to as the *secure world*. This is in contrast to the *normal world*, which is the main execution environment of the platform. The normal world usually contains the OS, drivers and untrusted applications, whilst the secure world contains only trusted code used by the applications. The TrustZone extensions provide a well-defined mechanism for code in the normal world to call functions provided by the secure world software. Since the separation between worlds is enforced in hardware, even an adversary with full control of the normal world cannot subvert this isolation. TrustZone technology is available on most modern smartphones, but has arguably not been used to its full potential [30].

2.3. Software Guard Extensions (SGX)

A more recent example of a technology that provides a hardware-enforced TEE is Intel's Software Guard Extensions (SGX) technology. An overview of the SGX protection model was given by Mckeen et al. [23]. In their paper, they present the core of this technology, the extensions that enable instantiating a protected container, describe the SGX instruction set, security model, threat model, and the hardware components on which this technology is based. In this section, we give a brief introduction to SGX and the protection capabilities it provides, which are relevant to this work.

- **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 Enclave Page Cache (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.

2.3.1. SGX Enclave Instructions and Protection Rings. The enclave instructions available with SGX are divided into two protection rings; ring 0 and ring 3 [31]. The allowed set of instructions is determined according to the privilege level of the executing software. For the most part, ring 0 instructions; ECREATE, EADD, and EINIT are used for EPC management and thus executed by privileged software such as OS and VMM, while ring 3 instructions e.g. EENTER, EEXIT, EGETKEY, EREPORT, and ERESUME are used by the user-space software to execute functionality within or between enclaves.

DOI: 10.1002/cpe

2.3.2. Enclave Life Cycle. In order to provide strong security features, managing an enclave is done in hardware through enclave build instructions. To create an enclave, the ECREATE instruction is used. It builds the enclave and sets base and range addresses. Once an enclave is created, EADD is used to add 4KB protected pages of data and code. This is followed by measuring the enclave's content using EEXTEND to protect the integrity of the data within the enclave. To elaborate on the latter, adding and measuring the enclave's pages are done by software prior to EINIT instruction. Once called, it finalises the measurement of the enclave and establishes an enclave identity. Executing within an enclave prior to this instruction is not allowed. On the success of EINIT, entry to the enclave is enabled and permitted to run on the processor in a privileged mode called *enclave mode*.

In order to enter and exit the enclave under program control, EENTER and EEXIT are used respectively. On enclave entry, the cached addresses are flushed, including addresses that overlap with the addresses used by the enclave to ensure the protection of the memory accesses within the enclave. Similarly, on enclave exit any cached addresses referring to the protected space in an enclave are cleared. The purpose of this is to prevent external software from using the cached addresses to access the enclave's protected memory.

2.3.3. Asynchronous Exit and Resuming Execution. An asynchronous exit from the enclave occurs due to events such as exceptions and interrupts, in which the processor handles such events by invoking the internal routine Asynchronous Exit (AEX). The AEX saves the registers used by the enclave, which are consequently cleared to prevent secrets leaking. In particular, one saved address to be stored is the location of the returning address, also called the faulting address, where the execution resumes on resuming the enclave's execution. While saving the enclave's state is essential for resuming the enclave's execution, equally important is clearing the data used by the enclave to prevent secret exposure. Once AEX finishes execution, the processor exits enclave mode and goes back to normal mode where every instruction is treated as an external instruction.

On the other hand, the ERESUME instruction restores the enclave's state and gives back control to the enclave from the point it was interrupted. It is important to mention that the event which the AEX was called upon may be triggered again in case of failure when the event is an exception or fault within the enclave.

2.3.4. Application Partitioning using SGX. An application uses SGX to protect the execution of sensitive portions of code by executing this code inside one or more enclaves. It is important to note that porting the code to run in an enclave is not the only action required when partitioning the code - the ported code should be able to handle I/O operations and external operations and exit enclave mode when necessary. The interface to the enclave is limited and the creation process requires the intervention of privileged software that runs in ring 0, e.g. an SGX driver. As a rule, the privileged software creates an enclave using ECREATE, then adds and measures the code of the desired partition. It uses EADD and EEXTEND respectively to perform these operations, and then performs EINIT to finalise the creation process. In order to enter an enclave, the application uses the synchronous entry instruction EENTER to switch the processor to enclave mode and to execute the relevant call. As an essential part of the design, I/O operations are excluded from the enclave since they require the intervention of the OS. Thus, when I/O operations are required, a synchronous exit (EEXIT) is called to switch the processor to normal mode to handle the requested external operation. In a similar way, OS interrupts are handled through *Asynchronous Exit and Resuming Execution instructions*. Once the interrupt has been completed, the application can resume the enclave's execution from a pre-defined re-entry point using the ERESUME instruction. Once the enclave finishes execution it exits enclave mode using EEXIT and the processor returns to normal mode of execution. The life-cycle of the enclave and its content can be terminated by the application using privileged software. The privileged software tears down the pages inside the enclave (using EREMOVE) and removes all the meta-data associated with the enclave.

DOI: 10.1002/cpe

3. ADVERSARY MODEL AND SECURITY OBJECTIVES

In this section, we specify the assumed capabilities of the adversary and define the security objectives with respect to this adversary.

3.1. Adversary Model

We consider a powerful adversary who has control over the platform. This adversary has the capability to run arbitrary software on the platform, read all platform memory, and manipulate the OS (including booting another OS). Our adversary model is inspired by previous work [32] [33]. 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. [34]).

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) [35] [36] [32] [33]. 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 Trusted Execution Environments (TEEs) 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. They are out of scope for our present purposes.

As explained in Section 2, 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 (as described in the previous section). However, our work exclude side-channel attacks and it is out of the scope of this paper.

3.2. Security Objectives

From the perspective of an application running on this type of untrusted platform, this 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.

DOI: 10.1002/cpe

- **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 all 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 TLS library, an example of data in use would be a session 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. Our security objectives 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. This therefore decreases 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 [24, 37].

It must be noted that we do not aim to ensure availability of security-sensitive code or data. 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 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.

4. APPLICATION PARTITIONING FRAMEWORK

We use the term *partitioning scheme* to refer 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 TEE. For any given application, there are multiple possible partitioning schemes, which 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 focused on a single application.

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 4.1. 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 our framework is that we can establish application-independent relationships between the different types of partitioning schemes.

DOI: 10.1002/cpe

These relationships can be used to guide the architectural design of partitioned applications. These relationships are explained in Section 4.2.

4.1. Types of Partitioning Schemes

As explained in the previous section, the purpose of this partitioning is to enhance the security of an application running on an untrusted platform. Our 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.

4.1.1. Type 1 - Whole Application In a type 1 scheme, as much of the application as possible 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.

4.1.2. 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 including only the security-sensitive code and data. As explained above, this also includes 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.

4.1.3. 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 could 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 by default. Unlike type 1 and 2 schemes, a type 3 scheme can only be implemented using technologies that provide multiple TEEs per application (e.g. 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*.

4.1.4. Type 4 - Hybrid A type 4 scheme refers to a combination of any of the preceding schemes. Multiple enclaves 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 enclave. 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 enclave, since compromise of either key would reveal the encrypted data. Similarly, an application with multiple users could have one enclave 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 enclaves in order to protect its integrity for individual users. As in type 3 schemes, a type 4 scheme requires multiple enclaves per application. Any type 4 schemes that require inter-enclave secure communication are referred to as *Type 4 Advanced*.

4.2. Relationships between Types

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

DOI: 10.1002/cpe

partitioning schemes defined above are application-independent, and can thus be qualitatively compared across applications in various dimensions.

4.2.1. 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.

$$num\_tees(T1) = num\_tees(T2) = 1; \;\; num\_tees(T3) \geq 2;$$
$$num\_tees(T3) \geq num\_tees(T4) \geq num\_tees(T2); \quad (1)$$

As explained above, both type 1 and type 2 schemes require only a single TEE. A type 3 scheme involves at least two TEEs[†]. 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.

4.2.2. TEE TCB Size. The size of the 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:

$$tcb\_size(T1) \geq tcb\_size(T2) \geq tcb\_size(T4) \geq tcb\_size(T3); \quad (2)$$

By definition, a type 2 scheme will have a smaller TCB than a type 1 scheme because the type 1 scheme aims to include as much of the application as possible, while the type 2 scheme only includes security-sensitive code and data. 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).

4.2.3. 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.

4.2.4. 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:

$$num\_entries(T3) \geq num\_entries(T4) \geq \{num\_entries(T1), num\_entries(T2)\}; \quad (3)$$

By definition, a type 3 scheme would have 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.

[†]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 schemes since they would be indistinguishable from type 2 schemes.

DOI: 10.1002/cpe

4.2.5. Summary. The qualitative relationships between the types of partitioning schemes are summarised in Table I.

Table I. 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	-	-	$\geq T2$	$\leq T3$

5. SECURITY AND EFFICIENCY EVALUATION

We use two case studies: a MiniServer + OpenSSL library, and the Apache web server, to examine several software partitioning schemes. 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 [38].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 [39]. Apache [40] 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 items 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 various enclaves requires trusted channels for communicating between these enclaves, thus increasing the complexity of the system and expanding the attack surface since there are more components to protect. On the efficiency side, we consider the number of enclaves, the number of entries to these enclaves, and the size of each enclave. Moreover, context switching is required when moving into and out of the enclave, introducing an overhead that increases with the number of enclaves and entries to these enclaves. Previous empirical work [38] investigated the ability of SGX to protect against exploitation of the Heartbleed vulnerability in OpenSSL with the four different partitioning schemes enabled. Based on the findings from the previous work, here we investigate some other vulnerabilities that are mitigated using our approach SGX enabled approach. The chosen vulnerabilities are from the same type class as the heartbleed vulnerability, memory + information leakage class as classified by CVEDetails [41]. We evaluate the security and efficiency of the proposed partitioning scheme types from section 4 and present the calculated results in table VI.

5.1. Case Study 1 - Apache Web Services

Apache holds today the biggest share of the web server market. Apache's success is due to its broad range of functionality, 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 these. 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 library like OpenSSL, ModSSL, and others. Unfortunately, these third party libraries might have vulnerabilities, when exploited might lead to information leakage or compromising the integrity of some parts of the importing application. An

DOI: 10.1002/cpe

example evident of that is the HeartBleed vulnerability in the OpenSSL that resulted in leakage of user-names and passwords in many applications such as Apache [42].

Apache has three main security modules [43], authentication, access control, and network communication using Secure Sockets Layer/Transport Layer Security (SSL/TLS). The authentication modules allow identification of clients, usually through verifying user-name and password 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.

5.1.1. Type 1 Scheme - Whole Application as One Partition. In the first scheme the entire Apache code and data resides in a single enclave. In this partition, even possible use of the OpenSSL by apache, we do not consider the OpenSSL library part of the partition inside the enclave. 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 user-names and passwords, can not be extracted/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 an enclave, encrypt the data in memory with a key not accessed to OpenSSL, hence, in case of buffer over-read allowed due to being in the same process, only encrypted data is read.

5.1.2. Type 2 Scheme - Single TEE. In the second scheme we use one enclave 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 vulnerability residing in other modules CVE-2014-3583,CVE-2010-2227 [41] since they can not access the data and code which is encrypted in memory as part of an enclave. 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.

5.1.3. Type 3 Scheme - Individual TEEs. In the third scheme we use multiple enclaves to isolate the different modules, and 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 of 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 to the original copy of the module code and data. In the same way modules are isolated with enclaves and enclaves 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 [41]. In addition, the 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.

5.1.4. Type 4 Scheme - Hybrid. In the fourth scheme we use multiple enclaves to isolate the different modules, and rest of the code used in Apache. However, unlike the type 3 scheme, we reduce the number of enclaves 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 enclaves used in order to reduce the overhead

DOI: 10.1002/cpe

Table II. 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 III. Apache Vulnerability Mitigated with the Four Partitioning Schemes

	Grade	First Scheme	Second Scheme	Third Scheme	Fourth Scheme
	Buffer Overflow				
CVE-2015-1836 [41]	7.5	✗	✗	✓	✓
CVE-2014-3583 [41]	5.0	✗	✓	✓	✓
CVE-2014-0226 [41]	6.8	✗	✓	✓	✓
CVE-2014-0085 [41]	2.1	✗	✓	✓	✓
CVE-2012-3502 [41]	4.3	✗	✗	✓	✓
CVE-2010-2227 [41]	6.4	✗	✓	✓	✓

switching between the environments. We call for this decision due to the characteristics of the code inside of the enclave, but this is out of the scope of this paper and will be addressed in future work.

The type 4 scheme, like the type 2, scheme protects against exploitation of external software such as the HeartBleed vulnerability or vulnerability 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.

Table II presents some of the Apache vulnerabilities as reported by the CVE Security Vulnerabilities Datasource [41]. 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 III presents the ability of each of the partitioning scheme of our solution to mitigate the vulnerabilities from table II.

5.2. Case Study 2 - OpenSSL

In this section we use the OpenSSL library to examine the proposed software partitioning schemes. Motivated by our previous work [38],we choose to mitigate vulnerabilities from the memory + information leakage class, the *HeartBleed* vulnerability [44], 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.

Figure 1 presents an attack exploiting the HeartBleed vulnerability between a client and a server.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

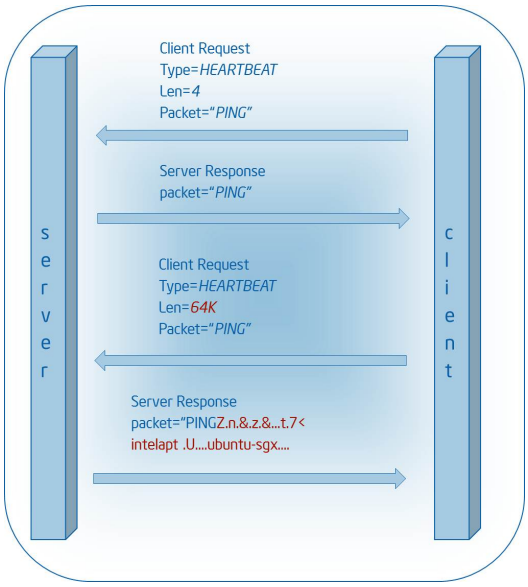

Figure 1. HeartBleed Example

The vulnerability known as HeartBleed results from a missing bounds check in the heart beat 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 and sensitive data. The sensitive data is not limited to secret keys used within the OpenSSL library, but also includes user names 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.

5.2.1. Type 1 Scheme - Whole Application as One Partition. In the first scheme the entire OpenSSL library resides in a single enclave and includes the heart beat code. The code within an enclave has memory access to every memory address inside the same enclave, thus when a client requests more data than it has sent, the heart beat 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 user-names 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 enclave. To illustrate this using the HeartBleed example, the heart beat code resides within an enclave, 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 enclave are transparent to the contained software, and accessing secrets from an inner function, such as the heart beat code, can be achieved without the enclave's interference.

Scheme 1 uses one enclave 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 it doesn't protect the confidentiality of secrets upon implementation.

5.2.2. Type 2 Scheme - Single TEE. In the second scheme we used one enclave to isolate part of the OpenSSL library including the handshake protocol, private key, and session key. We partition

DOI: 10.1002/cpe

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

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

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

In scheme 3 isolating each secret in a different enclave protects against code vulnerabilities, such as HeartBleed, compromising the confidentiality or integrity of the session key or private key. The TCB in each of the enclaves is significantly smaller than in scheme 1 . However, this approach brings with it other challenges: In order to prevent malicious software from exploiting the different enclaves, a trusted channel must be established between the different enclaves 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 paper.

5.2.4. 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 enclave. The heart beat code resides in the untrusted space of the application and is thus unable to access the secrets within the enclave. The heart beat code could reside in a separate enclave 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 scheme 1 and isolation between the sessions is achieved. However, TCB is not as small as in scheme 3. The advantage of scheme 4 over scheme 3 is a reduction in the number of enclaves. The number of trusted channels required between different enclaves 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-overflow vulnerabilities such as HeartBleed. In addition, with this scheme the size of the TCB inside the enclave proved to be much smaller than scheme 1.

Table IV present some of the OpenSSL vulnerabilities as reported by the CVE Security Vulnerabilities Datasource [41]. Table V present the ability of each of the partitioning scheme of our solution in mitigating the vulnerabilities from table IV.

Table IV. 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

DOI: 10.1002/cpe

Table V. 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 VI we summarise the analysis of the 4 different partition schemes discussed in previous section and previous work [38].

Table VI. Comparison between the 4 schemes - OpenSSL [38]

	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. RELATED WORK

In the last decade the topic of executing sensitive code in isolated and trusted environment has caught the attention of many researchers. McCune et al. presented Flicker [13], an infrastructure for code execution in isolated and trusted environment. In their work they rely merely on 250 lines of code in the TCB to provide strong isolation. For the most part, they appreciate that 250 lines of code is a tiny code, therefore formal assurance of its execution is more trusted as a result of the feasibility of verifying the code. Nonetheless, an application running in an isolated execution environment can be thousands of line of code and isolation between several parts in the application space is essential to prevent exploits by unfortunate vulnerabilities. The same group presented TrustVisor [14] a pointed purpose hypervisor that provides code and data integrity and secrecy for sensitive portions of an application. TrustVisor provides application developers with a strong secure environment for code execution and data storage on untrusted platforms. Moreover, they argue that small TCB code is easier to be formally verified, thus, it is more trusted when executing in TEE.

Another research effort that takes a similar approach is that of Singaravelu et al. [19] 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. suggests the use of dynamic libraries, commonly used in software development, as a mean for disaggregation [29]. In his work he propose dividing the software using an existing development technique to one or more dynamic libraries. In turn, a privilege software such as a kernel or a VMM enforces isolation between the two domains, the host process and the shadow

Prepared using cpeauth.cls
DOI: 10.1002/cpe

process that includes the dynamic libraries. Their approach uses the Xen hypervisor for coarse-grained isolation between the two domains, that is similar to scheme 2 in our proposed framework.

An alternative method to using isolation to protect secrets and mitigate the exploitation of vulnerabilities is through monitoring the execution of the software. Geneiatakos et al. and colleges suggest using virtual application partitioning to dynamically adapt the software defences based on the current execution partition of the application [45]. 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 partition of the application. For example, for data considered 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 [46]. 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 an essential need for another layer of protection, such as the one proposed in this work, to mitigate the impact of exploits. In another output from the same group [47]; 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.

Many pieces of research proposed Sandboxing to achieve security and privacy. In Robusta [48], the authors propose framework that provides security to native code in Java applications. Robusta, isolates native code and prevent external modification of the data and keeps the data 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 JVM to intermediate between the JVM and the native code.

In another work, Dune: safe user-level access to privileged CPU features [49]. The authors present a design that exposes the CPU virtualisation feature of Intel CPU to user applications. Instead of using the virtualisation for virtual machines as used in many systems, they use virtualisaion for a process which is much lighter weight than a virtual machine, 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 protections rings for processes, the privileged rings are exposed to normal processes which allow a process to run in a ring 0 privilege. Also, instead of using system calls it uses hyper calls.

In the mobile devices domain, the hardware architecture of Arm TrustZone perform separation between the "secure world" and the "normal world". However, it is very unlikely to get the approval of the OEMs 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 [50], 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 save the status of the executing software (the Rich OS) , configure 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 [51] [52] [53] [54]. These approaches are orthogonal to our approach and may rely on our approach to mitigate information leakage vulnerabilities. In our work, we rely on TEEs 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 [51] [52] [55] [56] explain attacks to obtain users' data and recognise that

DOI: 10.1002/cpe

these attacks wouldn't have been possible if the data were encrypted. Our partitioning approach using TEEs provides an efficient solution to this challenge.

In the network application domain, Kim et al. use OpenSGX, an emulator of SGX to solve the privacy and security issues in Software Defined Network (SDN) based inter domain routing and peer-to-peer anonymity networks (Tor) [57]. 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) [27], the authors present a tool for evaluating proposed compartmentalisations. The tool uses the annotation of the code that sandboxing boundaries of the code of a compartment, sensitive data such as keys that are not allowed to leave a compartment, past vulnerabilities as measure to the effectiveness of the applied compartmentalization. In SOAAP, the authors use compartmentalisations patterns [58] 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 rights to mitigate exploitation of the vulnerabilities.

Kilpatrick et al. propose a library that simplifies partitioning applications for privileged UNIX daemons. Privman require 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 operation 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 [59].

In [60] 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 [61], Cheng et al. presented DriverGuard, 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 [10] 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 [17].

In [25] 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 paper [26].

Partitioning privileges between hardware and software is not a new paradigm [62]. Hardware/Software partitioning has shown improvement in performance, energy consumption, and optimised run-time.

DOI: 10.1002/cpe

Table VII. Partition Isolation Methods Classification to the Four Partitioning Schemes

	First Scheme	Second Scheme	Third Scheme	Fourth Scheme
Thwarting Memory Disclosure [26]	✗	✓	✗	✗
Flicker [13]	✓	✗	✗	✗
TrustVisor [14]	✗	✓	✗	✗
Multilevel Security [63]	✗	✗	✓	✗
Xen [64]	✗	✓	✗	✗
Microkernel [65]	✗	✓	✗	✗
Chrome OS [66]	✗	✗	✗	✓
Partitioning Using SGX [38]	✗	✗	✗	✓
Privman [59]	✗	✓	✗	✗
Wedge [67]	✗	✗	✗	✓
Codejail [68]	✗	✓	✗	✗
Dune [49]	✗	✗	✓	✗

In table VII we classify related work according to our proposed partitioning schemes. Some systems enforce isolation between the different partitions through privileged software [14, 59, 64–67]. Other systems use virtualisation [49], partitions as linked libraries [68], processor hardware assistant [13], and trusted computing [14]. We classify these systems according to the granularity of the partitioning, for example a system that isolates an entire application such as Flicker [13] fits into the first partition 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 the case for Xen [64] and Privman [59]. However, none of these systems provided isolation per user/account/session. In the third partition schemes, applications are partitioned in a way to isolate the execution separately for each user/account/session. For example, in Wedge [67] 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 partition 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 founding to reduce the number of enclaves, thus, reducing the overhead produced of managing the enclaves. In scheme four, the partitions are optimised to reduce the overhead.

Our approach 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.

7. FUTURE WORK

In this work we presented a partitioning framework using TEEs to prevent exploitation of information leakage vulnerabilities. However, we did not investigate the ability of TEE technologies, such as Intel SGX or Arm TrustZone, to mitigate vulnerabilities other than memory leakage. For instance, TEEs can be used to reduce the impact of code injection attacks that attempt to steal application's data, such the case for inaudible data attacks [69] [33] [70] [71], or exfiltrate data residing in another TEE. However, although we experimented with information leakage resulting from exploiting a code injection vulnerability, we did not perform a definitive study on code

Prepared using cpeauth.cls
DOI: 10.1002/cpe

injection vulnerabilities. SGX enclaves prevent binary modification of the code inside the enclave because the code is measured prior to execution. We plan to investigate methods of using Intel SGX to reduce the impact of code injection attacks and other classes of vulnerabilities based on the adversary model in [69] [33]. Also, most TEEs today do not take into account different types of compartments, with different privileges. This topic is discussed in a previous work on compartmentalisation, where different compartmentalisation patterns [58] are presented to isolate software with different privileges. This approach has been discussed mainly when an Operating System is enforcing the isolation, but not in a hardware context for a technology like SGX.

In future work, we would like to propose methods that can mitigate vulnerabilities that are not memory related only using SGX. This can be achieved when combined with the other work we would like to address on compartmentalisation. We would like to investigate how to define compartmentalisation for technologies such as SGX, in order to enable a wider variety of options to target vulnerabilities and threats.

8. CONCLUSION

In order to protect sensitive code and data, it is desirable to use a trusted execution environment that is isolated from untrusted software, such as the OS. This can be achieved by keeping the TCB as small as possible and excluding irrelevant parts of the code. Fine-grained software partitioning of the code provides a good means of isolating different parts of the application and defining trust relationships 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 paper, we have proposed a framework that describes exactly *how* these technologies could be used to achieve secure software partitioning. We have defined four types of partitioning schemes that differ in terms of security guarantees and performance. We have shown that even though the partitioning schemes themselves are application-specific, it is possible to establish application-independent relationships between the different *types* of partitioning schemes. It is envisaged that this framework can be used by software architects and developers when designing partitioning schemes for applications that will make use of TEEs. We have demonstrated how each of these types of schemes could be realised using SGX to secure the execution of low level sensitive code in two case studies, the Apache web server and the OpenSSL library, as a proof of concept.

Although the TEE is an important and desirable security feature, it is not a silver bullet against vulnerabilities in code. We demonstrate a logical use of TEEs and the feasibility of designing different software partitioning schemes with SGX, even within a single application. In future work, we plan to perform broader research on fine-grained software partitioning using SGX with different applications, as well as benchmarking each of the types of schemes described above. Eventually, we intend to develop a methodology to help architects and developers to partition applications effectively using these new technologies in order to balance performance and security.

ACKNOWLEDGEMENT

The authors would like to thank the editor and the anonymous reviewers for their constructive and generous feedback.

REFERENCES

1. Misra SC, Bhavsar VC. Relationships Between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality. *Proceedings of the International Conference on Computational Science and Its Applications ICCSA 2003*, *Lecture Notes in Computer Science*, vol. 2667, Springer: Berlin, Heidelberg, 2003; 724–732.
2. One A. Smashing the stack for fun and profit. *Phrack magazine* 1996; **7**(49):14–16.

DOI: 10.1002/cpe

3. Sullivan N. Staying Ahead of OpenSSL Vulnerabilities — CloudFlare Blog [09 September 2014]. URL `http://blog.cloudflare.com/staying-ahead-of-openssl-vulnerabilities`.
4. England P, Lampson B, Manferdelli J, Peinado M, Willman B. A Trusted Open Platform. *IEEE Computer* Jul 2003; **36**(7):55–62, doi:10.1109/MC.2003.1212691.
5. Chen X, Garfinkel T, Lewis EC, Subrahmanyam P, Waldspurger CA, Boneh D, Dwoskin J, Ports DR. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM: New York, NY, USA, 2008; 2–13.
6. Martignoni L, Poosankam P, Zaharia M, Han J, McCamant S, Song D, Paxson V, Perrig A, Shenker S, Stoica I. Cloud Terminal: Secure Access to Sensitive Applications from Untrusted Systems. *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 2012; 14–14.
7. Garfinkel T, Pfaff B, Chow J, Rosenblum M, Boneh D. Terra: A Virtual Machine-based Platform for Trusted Computing. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, 2003; 193–206.
8. Ta-Min R, Litty L, Lie D. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006; 279–292.
9. Paverd AJ, Martin AP. Hardware Security for Device Authentication in the Smart Grid. *Smart Grid Security: First International Workshop, SmartGridSec 2012, Berlin, Germany, December 3, 2012, Revised Selected Papers*, Springer: Berlin, Heidelberg, 2013; 72–84.
10. Li Y, McCune J, Newsome J, Perrig A, Baker B, Drewry W. MiniBox: A Two-way Sandbox for x86 Native Code. *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, 2014; 409–420.
11. Hofmann OS, Kim S, Dunn AM, Lee MZ, Witchel E. InkTag: Secure Applications on an Untrusted Operating System. *SIGPLAN Not.* Mar 2013; **48**(4):265–278.
12. Atamli AW, Martin A. Threat-Based Security Analysis for the Internet of Things. *Proceedings of the 2014 International Workshop on Secure Internet of Things*, SIOT '14, IEEE Computer Society: Washington, DC, USA, 2014; 35–43.
13. McCune JM, Parno BJ, Perrig A, Reiter MK, Isozaki H. Flicker: An execution infrastructure for tcb minimization. *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, ACM: New York, NY, USA, 2008; 315–328.
14. McCune JM, Li Y, Qu N, Zhou Z, Datta A, Gligor V, Perrig A. TrustVisor: Efficient TCB Reduction and Attestation. *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010; 143–158.
15. Azab AM, Ning P, Zhang X. SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011; 375–388.
16. Sahita R, Warrier U, Dewan P. Protecting Critical Applications on Mobile Platforms. *Intel Technology Journal* 2009; **13**(2).
17. Yee B, Sehr D, Dardyk G, Chen JB, Muth R, Ormandy T, Okasaka S, Narula N, Fullagar N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, 2009; 79–93.
18. Dewan P, Durham D, Khosravi H, Long M, Nagabhushan G. A Hypervisor-based System for Protecting Software Runtime Memory and Persistent Storage. *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, 2008; 828–835.
19. Singaravelu L, Pu C, Härtig H, Helmuth C. Reducing tcb complexity for security-sensitive applications: Three case studies. *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, ACM: New York, NY, USA, 2006; 161–174.
20. Cheng Y, Ding X, Deng R. AppShield: Protecting Applications against Untrusted Operating System. *Singaport Management University Technical Report, SMU-SIS-13* 2013; **101**.
21. ARM. ARM TrustZone. URL `http://www.arm.com/products/processors/technologies/trustzone/index.php`.
22. McCune JM, Parno B, Perrig A, Reiter MK, Seshadri A. How Low Can You Go?: Recommendations for Hardware-supported Minimal TCB Code Execution. *SIGARCH Comput. Archit. News* Mar 2008; **36**(1):14–25.
23. McKeen F, Alexandrovich I, Berenzon A, Rozas CV, Shafi H, Shanbhogue V, Savagaonkar UR. Innovative Instructions and Software Model for Isolated Execution. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, ACM: New York, NY, USA, 2013; 10:1–10:1.
24. Hoekstra M, Lal R, Pappachan P, Phegade V, Del Cuvillo J. Using Innovative Instructions to Create Trustworthy Software Solutions. *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ACM: New York, NY, USA, 2013; 11:1–11:1.
25. Vasiliadis G, Athanasopoulos E, Polychronakis M, Ioannidis S. PixelVault: Using GPUs for Securing Cryptographic Operations. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014; 1131–1142.
26. Liu Y, Zhou T, Chen K, Chen H, Xia Y. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015; 1607–1619.
27. Gudka K, Watson RN, Anderson J, Chisnall D, Davis B, Laurie B, Marinos I, Neumann PG, Richardson A. Clean Application Compartmentalization with SOAAP. *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015; 1016–1031.
28. Saltzer JH, Schroeder MD. The protection of information in computer systems. *Proceedings of the IEEE* 1975; **63**(9):1278–1308, doi:10.1109/PROC.1975.9939.

DOI: 10.1002/cpe

29. Murray DG, Hand S. Privilege Separation Made Easy: Trusting Small Libraries Not Big Processes. *Proceedings of the 1st European Workshop on System Security*, EUROSEC '08, 2008; 40–46.
30. Ekberg JE, Kostiainen K, Asokan N. The Untapped Potential of Trusted Execution Environments on Mobile Devices. *IEEE Security & Privacy* July 2014; **12**(4):29–37, doi:10.1109/MSP.2014.38.
31. Schroeder MD, Saltzer JH. A Hardware Architecture for Implementing Protection Rings. *Commun. ACM* Mar 1972; **15**(3):157–170.
32. Azfar A, Choo KKR, Liu L. An android social app forensics adversary model. *49th Hawaii International Conference on System Sciences (HICSS)*, IEEE Computer Society: Kauai, HI,USA, 2016; 5597–5606.
33. Do Q, Martini B, Choo KKR. Exfiltrating data from android devices. *Computers & Security* 2015; **48**:74–91, doi:http://dx.doi.org/10.1016/j.cose.2014.10.016.
34. Xu Y, Cui W, Peinado M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. *IEEE Symposium on Security and Privacy*, 2015; 640–656, doi:10.1109/SP.2015.45. URL `http://dx.doi.org/10.1109/SP.2015.45`.
35. D'Orazio C, Choo KKR. An adversary model to evaluate {DRM} protection of video contents on ios devices. *Computers & Security* 2016; **56**:94–110, doi:http://dx.doi.org/10.1016/j.cose.2015.06.009.
36. Do Q, Martini B, Choo KKR. A forensically sound adversary model for mobile devices. *PLoS ONE* 2015; **10**(9):1–15, doi:10.1371/journal.pone.0138449.
37. Baumann A, Peinado M, Hunt G. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 2015; **33**(3):8:1–8:26, doi:10.1145/2799647.
38. Reineh AA, Martin A. Securing Application with Software Partitioning: A Case Study using SGX. *11th International Conference, SecureComm*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 2015; 605–621.
39. OpenSSL Software Foundation. OpenSSL Library Version 1.0.2a [13 April 2014]. URL `https://www.openssl.org/source/openssl-1.0.2a.tar.gz`.
40. Apache Web Server. The Apache HTTP Server Project [13 April 2014]. URL `https://httpd.apache.org`.
41. CVE security vulnerabilities database [2016]. URL `https://www.cvedetails.com`.
42. Samantha Murphy Kelly SFAS Lorenzo Francheschi-Bicchierai, Wagner K. The heartbleed hit list: The passwords you need to change right now [09 April 2014]. URL `http://mashable.com/2014/04/09/heartbleed-bug-websites-affected/#BZzq5PAodaqQ`.
43. Daniel Lopez Ridruejo. Apache Overview HOWTO [10 October 2002]. URL `http://www.tldp.org/HOWTO/Apache-Overview-HOWTO.html`.
44. N Mehta and Codenomicon. The Heartbleed Bug [13 April 2014]. URL `http://heartbleed.com`.
45. Geneiatakis D, Portokalidis G, Kemerlis VP, Keromytis AD. Adaptive Defenses for Commodity Software Through Virtual Application Partitioning. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012; 133–144.
46. Prokhorenko V, Choo KKR, Ashman H. Web application protection techniques: A taxonomy. *Journal of Network and Computer Applications* 2016; **60**:95–112, doi:http://dx.doi.org/10.1016/j.jnca.2015.11.017.
47. Prokhorenko V, Choo KKR, Ashman H. Context-oriented web application protection model. *Applied Mathematics and Computation* 2016; **285**:59–78, doi:http://dx.doi.org/10.1016/j.amc.2016.03.026.
48. Siefers J, Tan G, Morrisett G. Robusta: Taming the Native Beast of the JVM. *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, 2010; 201–211.
49. Belay A, Bittau A, Mashtizadeh A, Terei D, Mazières D, Kozyrakis C. Dune: Safe User-level Access to Privileged CPU Features. *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012; 335–348.
50. Sun H, Sun K, Wang Y, Jing J, Wang H. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, 2015; 367–378.
51. Do Q, Martini B, Choo KKR. Enhancing user privacy on android mobile devices via permissions removal. *47th Hawaii International Conference on System Sciences (HICSS)*, IEEE Computer Society: Waikoloa, HI, USA, 2014; 5070–5079.
52. Do Q, Martini B, Choo KKR. Enforcing file system permissions on android external storage: Android file system permissions (afp) prototype and owncloud. *13th International Conference on Trust, Security and Privacy in Computing and Communications*, IEEE Computer Society: Beijing, China, 2014; 949–954.
53. Petracca G, Atamli-Reineh A, Sun Y, Grossklags J, Jaeger T. Aware: Controlling App Access to I/O Devices on Mobile Platforms. *CoRR* 2016; **abs/1604.02171**. URL `http://arxiv.org/abs/1604.02171`.
54. Petracca G, Sun Y, Atamli-Reineh A, Jaeger T. AuDroid: Preventing Attacks on Audio Channels in Mobile Devices. *Proceedings of the 2015 Annual Computer Security Applications Conference*, ACSAC'15, 2015.
55. Pokharel S, Choo KKR, Liu J. Mobile cloud security: An adversary model for lightweight browser security. *Computer Standards & Interfaces* 2017; **49**:71–78, doi:http://dx.doi.org/10.1016/j.csi.2016.09.002.
56. D'Orazio CJ, Lu R, Choo KKR, Vasilakos AV. A markov adversary model to detect vulnerable ios devices and vulnerabilities in ios apps. *Applied Mathematics and Computation* 2017; **293**:523–544, doi:http://dx.doi.org/10.1016/j.amc.2016.08.051.
57. Kim S, Shin Y, Ha J, Kim T, Han D. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, 2015; 7:1–7:7.
58. Watson RNM, Woodruff J, Neumann PG, Moore SW, Anderson J, Chisnall D, Dave N, Davis B, Gudka K, Laurie B, *et al.*. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. *2015 IEEE Symposium on Security and Privacy*, 2015; 20–37, doi:10.1109/SP.2015.9.
59. Kilpatrick D. Privman: A Library for Partitioning Applications. *USENIX Annual Technical Conference, FREENIX Track*, 2003.

DOI: 10.1002/cpe

60. Strackx R, Piessens F. Fides: Selectively Hardening Software Application Components Against Kernel-level or Process-level Malware. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012; 2–13.
61. Cheng Y, Ding X, Deng RH. DriverGuard: A Fine-Grained Protection on I/O Flows. *Computer Security ESORICS 2011. 16th European Symposium on Research in Computer Security*, ESORICS'11, 2011; 227–244.
62. Stitt G, Lysecky R, Vahid F. Dynamic Hardware/Software Partitioning: A First Approach. *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, 2003; 250–255.
63. The Department Of Defense. Multilevel Security in the Department Of Defense: The Basics [1 March 1995]. URL `ftp://ftp.leeds.ac.uk/pub/caddetc/mls-basics.txt`.
64. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.* Oct 2003; **37**(5):164–177.
65. Accetta M, Baron R, Bolosky W, Golub D, Rashid R, Tevanian A, Young M. Mach: A New Kernel Foundation for UNIX Development. *In Summer Conference Proceedings 1986*, USENIX Association: Pittsburg,USA, 1986; 93–112.
66. Barth A, Jackson C, Reis C, Team T, *et al.*. The security architecture of the Chromium browser 2008. URL `http://seclah.stanford.edu/websec/chromium`.
67. Bittau A, Marchenko P, Handley M, Karp B. Wedge: Splitting Applications into Reduced-privilege Compartments. *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, 2008; 309–322.
68. Wu Y, Sathyanarayan S, Yap RHC, Liang Z. Codejail: Application-Transparent isolation of libraries with tight program interactions. *Computer Security - ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, 2012. Proceedings*, Springer: Berlin, Heidelberg, 2012; 859–876.
69. OMalley SJ, Choo KKR. Bridging the air gap: Inaudible data exfiltration by insiders. *20th Americas Conference on Information Systems (AMCIS 2014)*, Social Science Electronic Publishing: Savannah, Georgia, USA, 2014; 7–10.
70. Do Q, Martini B, Choo KKR. Is the data on your wearable device secure? an android wear smartwatch case study. *Software: Practice and Experience* 2016; :n/a–n/adoi:10.1002/spe.2414. SPE-15-0220.R1.
71. DOrazio CJ, Choo KKR, Yang LT. Data exfiltration from internet of things devices: ios devices as case studies. *IEEE Internet of Things Journal* 2016; **PP**(99):1–1, doi:10.1109/JIOT.2016.2569094.

DOI: 10.1002/cpe