Küçük *et al. EURASIP Journal on Information Security* (2019) 2019:14
https://doi.org/10.1186/s13635-019-0091-5

EURASIP Journal on Information Security

RESEARCH Open Access

Managing confidentiality leaks through private algorithms on Software Guard eXtensions (SGX) enclaves

Minimised TCB on secret-code execution with Early Private Mode (EPM)

Kubilay Ahmet Küçük[1*], David Grawrock[2] and Andrew Martin[1]

Abstract

Many applications are built upon private algorithms, and executing them in untrusted, remote environments poses confidentiality issues. To some extent, these problems can be addressed by ensuring the use of secure hardware in the execution environment; however, an insecure software-stack can only provide limited algorithm secrecy. This paper aims to address this problem, by exploring the components of the Trusted Computing Base (TCB) in hardware-supported enclaves. First, we provide a taxonomy and give an extensive understanding of trade-offs during secure enclave development. Next, we present a case study on existing secret-code execution frameworks; which have bad TCB design due to processing secrets with commodity software in enclaves. This increased attack surface introduces additional footprints on memory that breaks the confidentiality guarantees; as a result, the private algorithms are leaked. Finally, we propose an alternative approach for remote secret-code execution of private algorithms. Our solution removes the potentially untrusted commodity software from the TCB and provides a minimal loader for secret-code execution. Based on our new enclave development paradigm, we demonstrate three industrial templates for cloud applications: ① computational power as a service, ② algorithm querying as a service, and ③ data querying as a service.

Keywords: Trusted Computing Base (TCB), Software Guard eXtensions (SGX) Enclave, Private Algorithms, Secret-Code Execution (SCE), Algorithm Owner (AO), Hardware Owner (HO), Data Owner (DO), Enclave Developer's (ED) Responsibilities, Side-Channels, Early Private Mode (EPM), Internal Enclave Functions (IEF), Public Internal Enclave Functions (PIEF), Serialised Secret Internal Enclave Functions (SSIEF)

1 Introduction

Managing trust in remote execution environments is an enduring challenge. This is due to the fact that privacy-sensitive data and private algorithms remain unprotected in remote computers. There are a number of reasons why owners of private algorithms may need to run their algorithms in an untrusted environment. This may occur when the Algorithm Owner (*AO*) requires the capabilities of a Hardware Owner (*HO*), for example, to gain larger computing power in the cloud. In this case, the cloud infrastructure provider would be compensated and should take responsibility for an algorithm and system security. An alternative reason to run a private algorithm in a remote environment would be for the benefit of the *HO*, for example, the distribution of an industrial software product to end-users. In this case, the *AO* would be compensated, and thus responsible for security. It may also be the case that the consumer, whether that be the *AO*, or the *HO*, is responsible for security.

1.1 Ownership taxonomy

We use the following taxonomy throughout the rest of this paper. A ***Hardware Owner*** (*HO*) refers to any entity managing its own computational system. An ***Algorithm Owner***

*Correspondence: kucuk@cs.ox.ac.uk
[1]Robert Hooke Building, Cyber Security Centre, Computer Science Department, University of Oxford, Oxford, UK
Full list of author information is available at the end of the article

(*AO*) refers to any entity who owns the intellectual property of the secret code. Cloud infrastructure providers offer computational power, and they maintain the hardware and the software stack. In a cloud environment, the *HO* is a threat against the secrecy of private algorithms of the customers. Similarly in DRM[1], a *HO* is considered to be the end-user who may threaten the private algorithms with a commercial interest. From the *AO*'s position, the *HO* is considered to be both the cloud-provider (executing server-side code) and the end-user (executing client-side code). In short, the *HO* may be one or both of the following entities:

- An entity who generates revenue by selling the computational power
- And/or, an entity that needs the private algorithm to compute a secret value in her environment.

For both reasons, the *HO* must offer a trustworthy confidentiality service - either procedurally or through technical means - in order to convince the *AO* to use its services. Otherwise, the *AO* must take countermeasures before releasing its private algorithm.

The *AO* can use obfuscation techniques to protect the code before sending it to an execution environment. However, obfuscation methods are open to reverse engineering. On the other hand, the *HO* must make an effort within confidentiality management to gain the trust of the *AO*. For example, the *HO* develops the software stack to offer isolated execution environments and it aims to provide a remotely verifiable trusted computing base (TCB).

A ***Data Owner*** (*DO*) refers to an entity who has privacy-sensitive inputs. In Section 7, we consider two cases: ① Between the *AO* and the *HO* and ② between the *AO* and the *DO* where the *DO* controls the *HO*. The case where the *HO* may collude with the *AO* against the *DO* is out of the scope of this paper.

1.2 Taxonomy of the private and public assets

The *AO* and the *DO* have a choice to keep their assets as private or public. An asset, either an algorithm or a data-set, might be weakly or strongly private/public. We summarised the meanings of these new concepts in Table 1. In the rest of this paper, we use *private* or *public* keywords for the assets which are desired to be *strongly* private or public. We consider the *weakly* private or public assets to be *not private* and *not public*. These concepts are defined as follows.

Weak-private assets An algorithm or a data-set may not be publicly accessible, and be private against any third-party entities. Suppose that the asset owner sends this piece of information to a cloud platform for a remote computation. We classify this case **Weakly Private**, as the cloud operator may access to the assets.

Strong-private assets A security mechanism in the cloud may protect the confidentiality of these assets. Suppose that the threat model of this mechanism considers a malicious cloud owner. We call the assets **Strongly Private**, as they are protected against the insider threats in the cloud.

A private asset, therefore, can be a strongly private asset if the execution model satisfies the confidentiality requirement. We describe our secret-code execution model in Section 6 which allows *Private Algorithms* and *Private Data* to be strongly private.

Weak-public assets An asset might be accessible, but it may be too complex to analyse or too big to process. If an asset does not give much evidence about its trustworthiness, we call it **Weakly public**.

Strong-public assets We often use the keyword *public* to refer to the inspectable and attestable assets. For example, a public algorithm refers to a piece of code that is ① open for inspection by members of public, and ② attestable through a trustworthy evidence. These are the **Strongly Public** assets.

1.3 Distinguishing the private data and the private algorithm

Much of the current research [9, 14, 31, 34, 41, 44, 48, 51] is to keep *data* confidential in a Trusted Execution Environment (TEE). The novelty of our work is that, in addition to the input values and constant values used in an algorithm, we are adding algorithm secrecy. Our aim is to keep additional information secret *how* and *where* these values are used.

A function in a computer programme may include multiple assets such as sub-methods, operators, and constant values. The *AO* defines the *order* and *types* of these assets in a function.

Table 1 Definition of the private and public modifiers

Called as	**Refers to**	Properties	Example
Private data or private algorithm	**Strongly private**	Secret and protected	A secrecy-critical application logic or data-set
Not private	**Weakly private**	Special but not protected	A code or data uploaded to today's cloud servers
Not public	**Weakly public**	Open but not trusted	A code or data; too complex or too big for inspection
Public data or public algorithm	**Strongly public**	Inspected and attestable	A code or data with an evidence of trustworthy measurement

We summarise these assets in the following expression (1) to distinguish the data and the algorithm. For the given function (f), the *AO* may not necessarily provide the data (*input* (x) and constant (c)). To point out the difference, the *AO* nevertheless defines the mathematical operations and the application logic. In a decentralised setting, one or more *DO*(s) can provide confidential values, such as the inputs and the constants. The *AO* would provide the secret behaviour of how the algorithm uses the given values.

1.4 The problem of secret-code execution (SCE) with private algorithms

The success of a commercial algorithm may be measured on how widely it is distributed. However, the integrity and confidentiality of this algorithm must also be ensured. The problem arises when either party cannot trust the other, whether that be the execution environment, or the product.

The problem is that the *AO* needs to run code on the *HO* and the *AO* does not want the *HO* to discover any details regarding the code.

In untrusted remote environments, a computation including commercially valuable algorithms - for example, feature engineering in machine learning, business logic, or financial applications - may pose confidentiality concerns. Ideally, an *AO* would keep their private algorithm in their own physical location. A decentralised setting, however, requires the *AO* to send their private algorithm to an execution environment that may be owned by another entity (e.g. the *DO* or the *HO*). A hospital holding secret data may require all computations to be performed in their environment (considered as the *DO* or *HO*), while the hospital environment is a threat against secrecy of the algorithm. If the *AO* uses a cloud service for more computational power, uploading the secret binary to cloud service may leak the application code due to reverse engineering. In another case, an authority may need to run a private algorithm on computers of end-users (considered as the *HO*), requiring security guarantees to be managed by the *AO* or the *HO*.

1.4.1 The security problem on hardware software composition

In order to solve the problem of private algorithms, both distrustful parties could utilise secure hardware enclaves. A fundamental question rises as to whether the *AO* should develop its enclave, or trust to the enclave developed by the *HO*.

In Section 2, we explain the *enclave* concept and its development model in detail. In short, the term enclave refers to the area of one entity that is surrounded by another entity. Intel introduced the enclave concept into the security world with their Trusted Execution Environment (TEE) on a new instruction set called SGX [23]. Enclaves are developed by an entity called *Enclave Developer ED*, explained in Section 2.2.

Developers may use the existing application binaries inside secure hardware enclaves with small or no porting effort. However, third-party packages programmed with no security in mind will surely bring security risks. TEEs cannot convert a non-secure application to a secure application. In the building-block approach to application design, even if each block is secure, the overall system may not be secure due to non-compositionality of security [16].

Practical TEEs [19] are currently the subject of widespread research, as their correct use and capabilities are not yet fully known. The aim of this paper is to answer the following research questions:

- How can we correctly utilise TEEs' security guarantees to protect private algorithms?
- What are the responsibilities of TEE application developers?
- What is the impact of the software TCB components to code confidentiality?

In this paper, we examine these questions in recent studies and in our solution. We show the challenges and responsibilities on the TCB design and its requirements for secret-code execution in a remote environment. We analyse the existing frameworks for confidentiality management, and we evaluate the attacks against code secrecy in the software stack. Finally, we present a new solution with a smaller TCB size and address a stronger adversary model.

1.5 Paper structure

Section 2 provides a background on current binary execution mechanisms, as well as responsibilities of application developers for the chosen development model. Section 3, explains the trade-offs between design choices in TCB for private algorithms. We analyse the TCB components of existing frameworks with practical attacks against code secrecy in Section 4 and Section 5 explains the bad practices in enclave design and development. Section 6 shows our design for secret code execution with reduced TCB that is providing better security. Finally, Section 7 demonstrates our method in practice with three use cases of private algorithms for industrial use.

$$function_f(input_x) = method_m(input_x)operator_o(constant_c) \tag{1}$$

Equation (1) Splitting the Algorithm and the Data in a function.

1.6 The motivation

The motivation for TCB minimisation comes from the secrecy guarantees that depend on a remote system's

TCB components. There are a number of ways in which developers can cause security problems in a system. It is common for developers to include third-party software in their enclave TCB. Unfortunately, they often fail to perform security and compliance analysis between the third-party package and the underlying hardware. Additionally, developers may fail to understand hardware and software co-design while constructing secure systems. Careless construction of composite parts within a TCB may also cause the loss of initial security guarantees of the hardware. This may be the main, sometimes initial, source of security problems. In this paper, we analyse two existing frameworks that address client-side secret code execution, but their TCB suffers from bad practices explained below. To solve these architectural design problems, we present a solution with a smaller TCB, and secure TCB composition for secret-code execution in remote environments.

We aim to eliminate these bad practices, summarising the motivation for this paper in three points:

- Increase of TCB size—The TCB size must always be minimal in order to avoid security risks, and enable possibility of formal verification.
- Weak software in the TCB—The third-party software packages included in a TCB must pass the security requirements of the all assets.
- Non-compositionality of security—Two secure components may not necessarily comprise a single secure composition. Even secure software in the TCB may create additional security issues due to composition problems with the underlying hardware. This may also void the hardware security guarantees.

1.7 Related work

In this section, we provide an overview of the studies on TCB minimisation and secure TCB design. Beyond the CPU and memory protections, Ports and Garfinkel demonstrate [43] the impact of malicious OS behaviour against the secure application design. Their arguments outline the attacks and mitigations via the trusted interface, and they explain utilisation of partitioning methods to reduce the TCB size. Singaravelu et al. [52] also demonstrated three case studies in which they were able to use kernelised TCB at OS security level on a commodity computer.

Similar to the arguments found within this paper, Piessens et al. [42] argue that developers have limited understanding of security guarantees offered by the execution infrastructure. Their work [42] focuses on language safety and full abstraction in programming language translations. McCune et al. [38] utilise TPM, AMD SVM, and Intel TXT to provide secure hardware primitives for minimised software TCB. Strackx et al. [56] discuss the use of memory-safe languages on protected module architectures for code and data security, and they explain how secure enclaves might be written. In our work, we focus on confidentiality issues due to the TCB components for secret-algorithms in a practical TEE called SGX.

Linn et al.'s work [37] in binary obfuscation aims to make reverse engineering difficult for application binaries. The work [60] from Wu et al. focuses on malware evasion, which hides the application code against the analysis. Similar techniques can partially protect the application code; however, they fail to maintain the algorithm secrecy in stronger adversarial assumptions. These cases include the decentralised settings where mutually distrustful parties are involved, and where the adversary has full physical access over the host platform. Even though binary obfuscation methods seem to have similar goals to our work, they remain in the scope of reverse engineering. Our work is in the domain of securing private algorithms executed in an untrusted cloud. The forensic tools can recover evidences from the memory about an operation. This information retrieval applies mainly to the data. Some forensic techniques can recover [1] the execution states of a known software. However, these techniques do not threaten the algorithm secrecy directly. We protect the private algorithms at runtime and provide protection before the execution.

A recent work called Golem Network[2] provides a decentralised marketplace for computational power. The golem enclaves are similar to the interpreter enclaves we discuss in this paper. The third-party libraries included in the TCB of interpreter enclaves may break the security guarantees offered by the SGX. The difference to our approach is that a Golem enclave has a large TCB size due to the Library OS (Graphene-ng) and the unmodified applications (e.g., Blender). With a larger TCB size, formal analysis becomes more difficult. Second, Golem Network provides data secrecy only, and inherited security guarantees might be limited, as we show in a case study on similar interpreter enclaves in Section 4. Similarly, in one of our industrial use cases, we present a template for computational power in Section 7. Our template does not use third-party libraries to process secrets, and it enables running algorithms strongly-private in remote computers.

The novelty of our work is as follows: ① we consider mutually distrustful entities (HO, DO, AO) with conflicting interests in the cloud, ② we differentiate the private algorithms and the private data, ③ we show the bad practices on use of TEE in the cloud, ④ we create a taxonomy for secure execution of private algorithms in untrusted remote environments, ⑤ we provide practical insights to enclave development, ⑥ we perform a security analysis on existing dynamic code loaders with interpreter enclaves, and ⑦ we evaluate our execution model in three adversarial settings in the cloud.

1.7.1 Research direction

The Horizon2020 funded research projects under SERECA[3] focus on a number of goals to build secure enclaves. These goals include application partitioning [36], trusted architectures for web services [8, 33], container architecture [3] and library support for unmodified applications (SGX-LKL[4]), better integrity [5] and isolation [7], and enclave memory safety [35] in the cloud.

European Commission funded several projects on TEE research under SecureCloud (TRUSTEE)[5]. The main direction of TRUSTEE projects is dependability in the cloud. In the context of SecureCloud, the *dependability*[6] notion includes confidentiality, but this applies to data only. The SecureCloud solutions [9, 14, 31, 44, 51] focus on processing confidential data via public algorithms in an untrusted cloud. There is no concern about keeping the algorithms secret in SecureCloud projects. The advantage of our work is that we keep the algorithms secret in the untrusted cloud. Both SERECA and SecureCloud projects consider the confidentiality and the privacy of the *data* only.

The nature of enclaves requires enclave code to be publicly known. The existing work focuses heavily on private data processing through those enclaves. The difference with this paper is that we use the publicly known enclaves to enable private algorithms in the cloud. We show two different approaches in Section 2.3: ① the *HO* develops an enclave that maintains the code-secrecy after its release, and ② the *AO* develops an enclave that ensures the code-secrecy before its release.

There are other projects worth mentioning listed under Intel SGX Academic Research[7] page. GrapheneSGX [58] provides library support for unmodified binaries. Projects [12, 13, 34, 41, 48] on privacy-preserving data-analysis provide data confidentiality. Ryoan [24] provides a two-way sandbox for enclaves. Moat [53] helps to formally verify the enclave code. AsyncShock [59] exploits the time-of-check-to-time-of-use (TOCTTOU) bugs. VC3 [45] from Microsoft provides both data and code confidentiality, but this applies to MapReduce functions only. Controlled-channel attacks [61] can leak the secret data. We analysed these attacks against the other frameworks [17, 20] executing private algorithms client-side in Section 4.

2 Background

Intel's SGX is a trusted hardware solution [2, 23, 39], which provides a novel development model for enclave binaries, as well as hardware-maintained (ring -3) integrity guarantees for computations at user-level (ring 3). It also provides trusted computing primitives such as *Root of Trust for Measurement* (RoTM) for its own execution, *Root of Trust for Reporting* (RoTR), *Root of Trust for Storage* (RoTS), and access control for multiple isolated memory regions.

The RoTM, at the lowest privilege level, generates trustworthy evidence about the state of a system [2]. In the case of SGX, RoTM provides evidence about the enclave's memory layout, not about the system outside of the enclave memory. SGX's enclave development model helps by securing sensitive parts of user-level applications [39]. Enclaves can prove their identity to verifier entities who require evidence before proceeding to execution. Therefore, SGX is a good TEE solution for application developers.

Other than the SGX instruction set, the SGX SDK includes SGX drivers, architectural enclaves and SGX trusted libraries. Intel actively improves the SDK and plans to add more instructions in SGX Version 2. SGX, together with SDK, is an ever-evolving software technology that provides pratical understanding of a TEE.

2.1 How universal is the enclave research?

This paper is in the domain of the Enclaves and TEEs. We currently use SGX-enabled hardware and we make our contributions with Intel SGX enclaves. However, enclave research goes beyond Intel's SGX hardware. ARM and AMD also provide TEE solutions. Microsoft recently announced[8] the OpenEnclave SDK which provides an abstraction to the underlying hardware.

An enclave programming model may be considered as independent from the SGX hardware. For example, a trusted hardware [15] may adopt Intel's SGX enclave programming model. Ideally, developers may replace the underlying hardware with an open hardware solution, providing stronger security guarantees.

2.2 How do SGX enclaves work?

The enclave programming model requires an Enclave Developer (*ED*) to programme, compile and trigger the binary into allocated enclave memory before execution [26]. In a decentralised setting, there could be three separate entities who program, access, and call the enclave binary. An issue may occur, however, if these entities are from different parties with conflicting interests.

A key feature of SGX hardware is that the CPU can measure the enclave memory [39]. This measurement represents the identity of the enclave. An enclave can contain any C functions except a few illegal instructions [25]. Enclaves can directly or indirectly communicate with the system, other applications, other enclaves, and other entities in the network. We give further information ① on enclave development with different SDKs in Section 3, and ② explain some good and bad practices on enclave TCB design in Section 5.

Other than splitting an application into ***Trusted*** and ***Untrusted*** components (partitioning), the trusted part

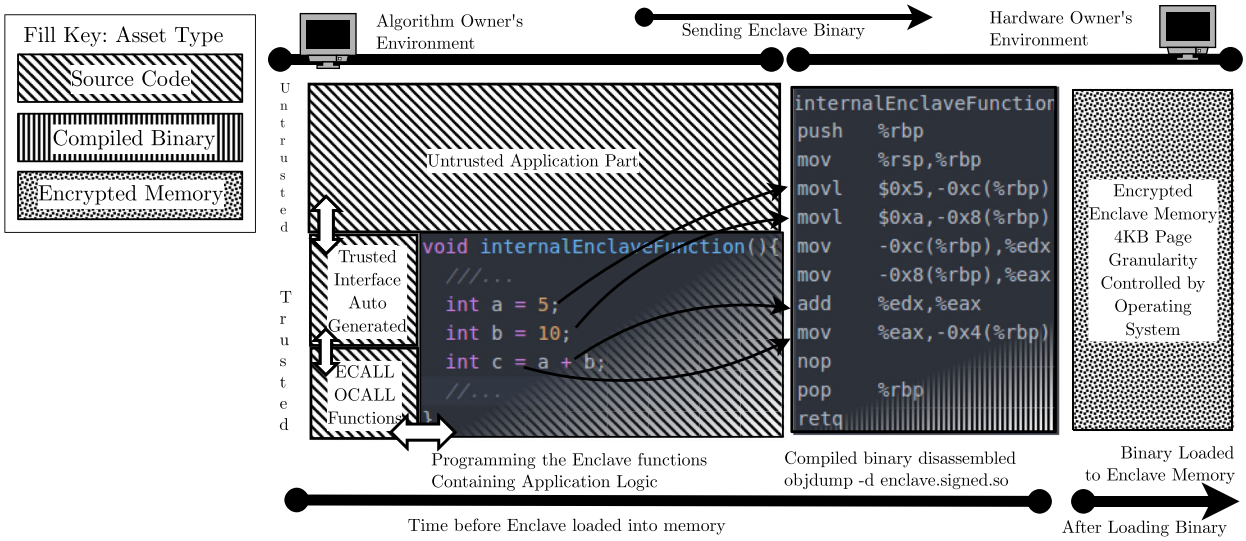

Fig. 1 Default Enclave Execution Mechanism of SGX between the Algorithm Owner (*AO*) and Hardware Owner (*HO*). The operator of the machine, the *HO*, can inspect the enclave code by reverse engineering the enclave binary. Enclaves must not include any secrets embedded in binary. Enclaves can, however, receive secrets through a secure channel or recover sealed data

(for example, enclave binary as a shared object; .so file in Linux) contains [28] two parts. These are the ***Internal Enclave Functions*** *(IEF)*, which contain application logic, and the ***Interface Functions***. Along with other parts of the enclave, the *IEF* are open for inspection before the initialisation process (Fig. 1). As is standard, if the *AO* (who places the application logic into enclave binary) and the *HO* (who calls the binary and executes the binary) are mutually distrustful, then the *HO* may gain information about the *IEF* and the algorithms compiled into the enclave binary.

2.3 What differs on protecting the private algorithms before releasing it or after receiving it?

There are two ways to secure private algorithms between parties with conflicting interests. The *AO* may secure the private algorithm via early operations on the algorithm prior to delivery. Alternatively, the *HO* may preserve the secrecy of the private algorithm after the delivery.

Both Approach 1 (protection before) and Approach 2 (protection after) (Fig. 2) have advantages and disadvantages in terms of usability and security. In Approach 2 (Fig. 2), the *HO* provides an enclave containing the dynamic code loader and execution method for the private algorithm. To provide this, the *ED* works with the *HO* to create a publicly known enclave for common use. The publicness of this enclave code is required, as the *HO* must convince the users to trust her enclave implementation. If this enclave code is not open-sourced, the *HO* can implement hidden functionalities. She will be free to signal the secret-code and the secret-data to herself through covert channels. In this case, the *AO* does not need to perform any extra operations on the private algorithm. This approach gives better usability to the *AO*, but it requires her to trust to the TCB designed by the *ED* and the *HO*. Section 4 evaluates the confidentiality management of secret-code handled by the *HO*, as is displayed in Approach 2 (Table 2). We classify the development types as follows: **Enclave-aware coding**: The developers create the algorithms to operate as the enclave code, with no intermediary layer. **Enclave-independent coding**: The developers create the algorithms for an intermediary layer (such as interpreters). This layer may or may not run in an enclave.

SGX hardware can help to preserve the confidentiality of the enclave applications against direct memory peeping and memory snooping attacks. Also, the *ED* can optionally implement the enclave with resilience to a set of side-channel attacks in order to address a stronger adversary model. To use an enclave for secret computation, developers may create publicly known enclaves with code-loading ability combined with interpreters, and load a secret-code on top of the interpreter to execute it. A weakly developed enclave may provide neither confidential data processing nor confidential execution of the secret-code.

3 SDK and TCB for the interpreter enclaves

Interpreter enclaves for private algorithms may consist of two parts: the dynamic secret-code loader mechanism and the script interpreter mechanism. The interpreter mechanism with rich functionalities may have system dependencies.

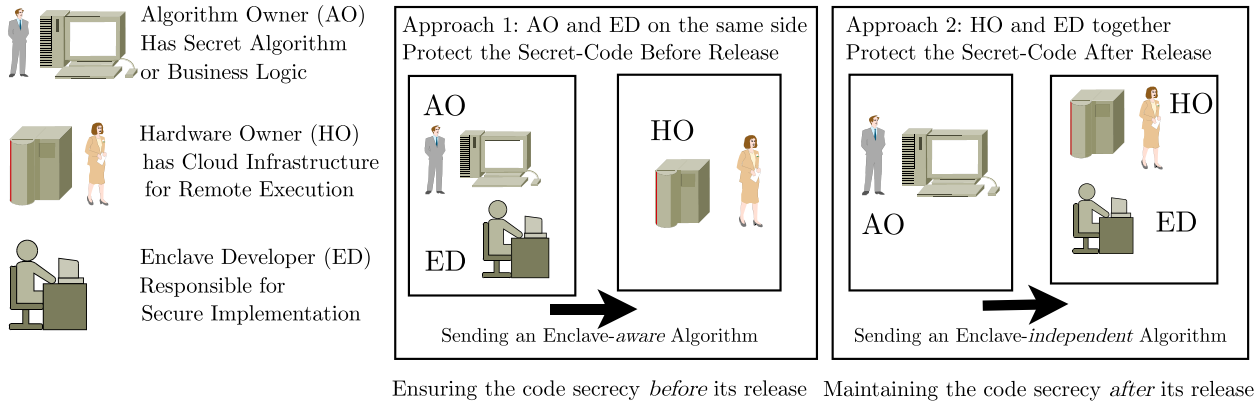

Fig. 2 This figure shows the three separate stakeholders in running private algorithms inside enclaves and two approaches to secret code protection. **Approach 1**: Enclave Developer (*ED*) works with the Algorithm Owner (*AO*) to protect the code within a secret part of the enclave. This requires the *AO* to perform early operations on the private algorithm before release. **Approach 2**: The *ED* works with the Hardware Owner (*HO*) to develop a publicly known interpreter enclave. This does not require the *AO* to perform early operations on the private algorithm

At the time of writing this paper, Intel SGX SDK only supported the use of C programming language to create an SGX enclave. A recent survey by Data Science [30] shows that researchers prefer Python and other languages over C and C++ for data analytics. In addition to this, web applications and computations on the client side widely use the Javascript language. To execute Python or Javascript code inside an enclave, the language interpreter must be embedded or ported to the enclave binary. A common way is that *ED* may prefer to port or adopt the open-sourced interpreters. To develop such an enclave, the *ED* can use the partitioning method using Intel SDK (recommended if no system library dependency is required). This can be achived by removing or replacing the illegal instructions and providing a trusted interface to the ported binary. In this method, the TCB size stays smaller in comparison to unmodified binaries (e.g., an enclave comprising an unmodified application code) supported with a Library Operating System (LibOS). If developers prefer running unmodified binaries that require system support inside an enclave, they must either use a shim layer [50] to forward the calls or embed the unmodified binary interpreter within a LibOS, (e.g., Graphene LibOS with Graphene SGX SDK). The Linux Kernel Library (LKL) is an alternative LibOS for this goal. Both of these LibOSs can support unmodified interpreters, but the act of using a LibOS increases the TCB size dramatically. The interpreters are able to handle complicated tasks and provide rich functionality; however, they require a LibOS support.

Table 2 Two approaches on *secret-code execution* through SGX enclaves

	Approach 1	Approach 2
Responsible for secrecy	Algorithm Owner	Hardware Owner
Secrecy ensured	Before sending the code	After receiving the code
Development type	Enclave-aware coding	Enclave-independent code
TCB/threat analysis	In Section 6	In Section 4
Enclave developer	The Private Algorithm must be developed for Enclave	The Enclave must be made for Private Algorithm

3.1 Understanding the TCB of enclaves

The SGX development model helps developers to create applications that, ideally, have a minimal TCB size. Because the larger TCB on a commodity computer may contain potential vulnerabilities, minimising the TCB reduces the risk of having vulnerable code. Developers are responsible for deciding what to include in the TCB of an enclave. If a developer includes an arbitrary code from third-party resources in an enclave, this may weaken or destroy the integrity guarantees, or fully destroy the integrity and confidentiality guarantees of the SGX hardware. The design and implementation of enclave (i.e., TCB) components are crucial for the security of the application. In other words, the underlying secure hardware may not protect the assets processed by the bad software stack. The co-design of hardware and software ensures proper management of the integrity and confidentiality guarantees.

3.2 Comparison of SDKs for enclaves

An application may require system calls to operate, or it may run entirely independently without any system dependencies. Depending on the requirements of an

application, there are three different ways of developing enclaves:

- Partitioning an application and using trusted interfaces within the enclave. We describe application partitioning with Intel SDK in 3.3.
- Using LibOS inside an enclave to support the dependencies. We describe development with LibOS with Graphene SDK [58] in 3.4.
- Using shim layers to filter system calls from enclave to outside world [50].

3.3 Developing enclaves with Intel SDK

Intel's SGX SDK requires developers to split an application into two parts: trusted and untrusted. Developers compile both parts into executable binaries. The untrusted application calls the trusted binary and maps it into a memory area allocated for the enclave. Now, the untrusted application can interact with the enclave via the trusted interface. The trusted interface then passes the data or requests to the internal enclave functions via ***E**nclave **CALL**s* (ECALLs). If any internal enclave function needs a system call, this request goes through the ***O**utside **CALL**s* (OCALLs) to the untrusted application in the outside world. If the internal enclave function does not require the use of system calls, it may perform all of its computations without OCALLs, further reducing the risk of a system call-based attack [11].

3.4 Developing enclaves on Graphene SDK

The Graphene SDK places the Graphene LibOS [58] (similar to the LKL[9]), into an enclave. In this setting, the enclave does not need to make any OCALLs for system dependencies to the untrusted operating system, as the Graphene LibOS can handle all necessary system calls. Graphene-supported enclaves can contain unmodified binaries. The amount of code inside an enclave, however, can extend to tens of thousands of line of code, making the process of formal verification very difficult. Both the larger enclave code and the larger TCB size can increase the risk of security vulnerabilities.

3.5 Private algorithms on SGX enclaves

Once an interpreter is either ported or embedded into an enclave, a dynamic loader is required to fetch, decrypt and load the code for execution. Figure 3 shows the feasibility of three approaches to deploy interpreters in enclaves based on their TCB components. The interpreter enclave would ideally have a small TCB size and support rich functionalities.

Loader + MuJS[10] JavaScript Interpreter + Intel SDK Developers can use a dynamic code loader at runtime to fetch Javascript code and decrypt the blob inside the enclave. MuJS interpreter ported for Intel SDK can interpret the loaded secret-code in the execution phase. This method provides a small TCB size compared to that which is used with LibOSes. *Enclave 1* within Fig. 3a shows the TCB components of this approach. It provides comparably minimal TCB size to that shown in Fig. 3b for an interpreter enclave. The *Enclave 1* is used [17, 20] by Goltzsche et al. and Fernandez et al., as analysed and evaluated in Section 4.

Loader + MuJS JavaScript Interpreter + Graphene SDK The second method (*Enclave 2* in Fig. 3) to create an interpreter enclave is to deploy MuJS on Graphene SDK. This method reduces the development effort because

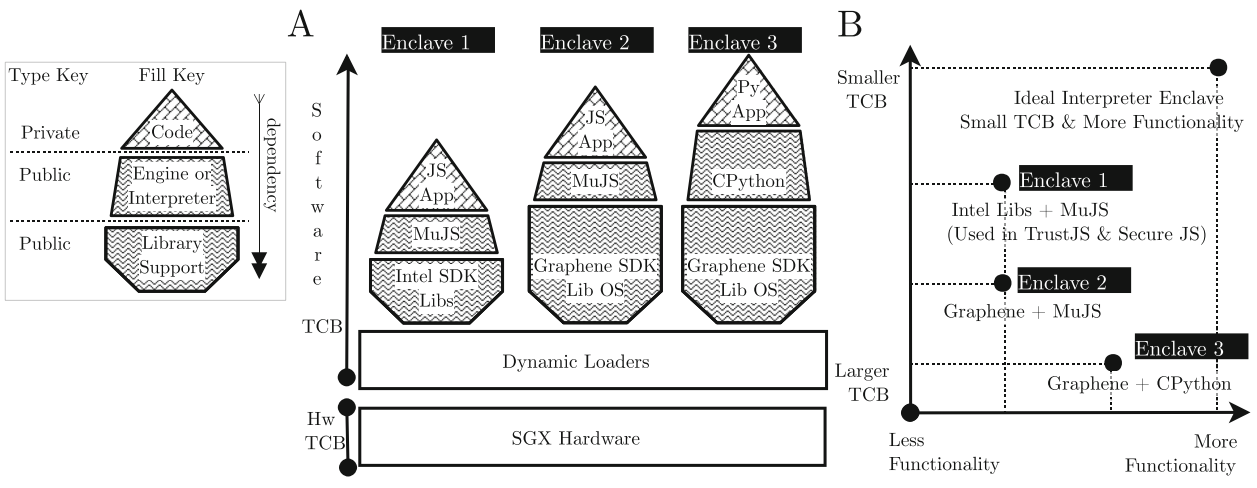

Fig. 3 Three different TCB to deploy Dynamic Code Loaders and Interpreter Enclaves for Private Algorithms. **a**) TCB Size may vary depending on SDK and interpreter of the language. Graphene SDK TCB > Intel SDK TCB. CPython TCB > MuJS TCB. **b**) TCB Size vs Functional Capabilities in design of Interpreter Enclaves. Ideally, the Interpreter Enclave will provide rich functionalities and will have a small TCB size

Graphene Library OS can support an unmodified MuJS interpreter, though it increases the TCB size. MuJS interpreter is smaller in comparison to CPython interpreter (while there are advanced JS engines, such as V8, MuJS is a lightweight option). As such, deploying a lightweight interpreter on a LibOS would not be an ideal model, because the majority of the TCB would not be in use.

Loader + CPython Python Interpreter + Graphene SDK The third method to deploy an interpreter enclave is to use Python, running over the Graphene LibOS, and to combine a dynamic code loader to fetch encrypted Python code at runtime and load it into the interpreter. This method can provide richer functionality for data analytics applications. The practical example of CPython Interpreter and Graphene SDK, without a dynamic secret-code loader, is available in the Graphene SGX SDK repository[11].

4 Case study: leaks on frameworks enabling confidential code execution

Both TrustJS [20] and SecureJS [17] frameworks enable a dynamic load of JS code at runtime. These frameworks use the first method described in Section 3.5. This method involves porting the MuJS for the Intel SDK, and creating a JavaScript *Interpreter Enclave*. Their performance results and a detailed comparison is available in this study [17]. However, direct porting of a third-party interpreter may not help to hide a private algorithm. According to the threat model of Intel SGX described in Section 5, the *ED* is responsible for the security of interpreter enclaves. We evaluate the attack surfaces and the software attack vectors in the following sections.

4.1 Attack surface on interpreter enclaves

The load and execution flow is the same in both frameworks of TrustJS and SecureJS. First, the Interpreter Enclave fetches the secret Javascript code in an encrypted blob. Inside the enclave memory area, MuJS needs to read the Javascript code as plaintext. The enclave dynamic loading mechanism prepares the received blob for the interpreter. Secondly, the interpreter parses each Javascript operation and calls the corresponding C functions. Frameworks use a custom application-specific code for the first phase of the preparation of the encrypted blob. Then, the MuJS interpreter (open-source) performs the second phase of execution.

TrustJS is not an open-sourced enclave[12], however, SecureJS was open-sourced[13] in June 2018. To analyse the code execution in SecureJS's TCB, we focus on debugging MuJS on Intel SDK displayed in Fig. 4, phase 3.

Attack vectors A straight-forward composition of ported third-party software packages on secure hardware enclaves introduces new attack vectors. Previously performed attacks [6, 21, 22, 40, 61] have shown how commodity software leaks secrets, while defence mechanisms explain how to mitigate these attacks on SGX enclaves. For this paper, we chose a non-trivial attack [61], performed with `data-dependent data access` and `data-dependent control flow` weaknesses in interpreter enclaves. If the enclave code has input-dependent control-transfer; e.g., calling different methods based on an input, the adversary can observe the called page externally and learn the private input. Similarly, if the data access pattern is based on an input parameter, the adversary can learn the private input.

Evaluation of TrustJS and SecureJS MuJS was not designed to be an oblivious interpreter, and its use in an enclave for secret-execution requires secure composition and further isolation techniques. The direct use of the unmodified binary of MuJS in an enclave in a security domain brings potential security threats. We have identified two examples of weaknesses in MuJS used in TrustJS and SecureJS frameworks. These frameworks rely on the implementation of the MuJS interpreter.

Along with the interpreter, the dynamic code loader can be another target for secret code leaks. By design, the dynamic code loader must be public, and therefore open for inspection. This is because, as explained in

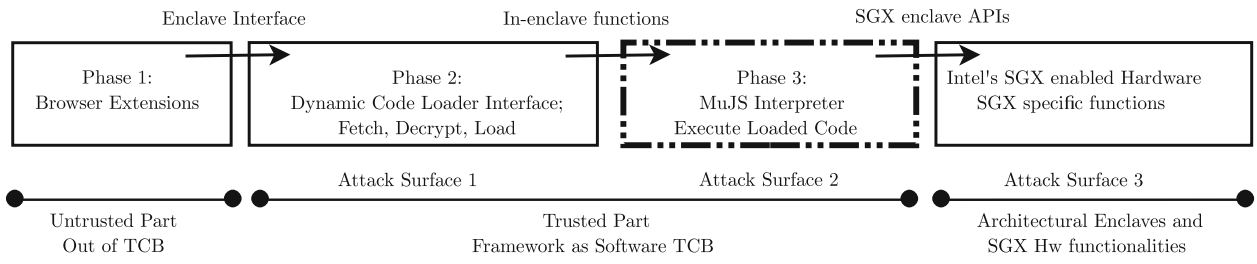

Fig. 4 Attack surfaces on TrustJS and SecureJS in two phases (2 and 3). Phase 1: Browser extension receives the code. Phase 2: Dynamic code loaders prepare the code blob. Phase 3: Interpreter executes the code. The confidential code may leak due to attacks placed on surface 1 and 2, targeting the weakly developed enclave

Section 2.3, the *HO* (the platform owner, the cloud owner, the infrastructure provider) has to convince the public that her enclave does not perform any dishonest operations. Open-sourcing the interpreter enclave is in the interest of the *HO*. The *HO* can increase her profit in this type of cloud scenario, if she can convince more users to upload their secrets.

In this case study, the *AO* needs to attest the public code loaders. The attestation guarantee, however, would only prove that a known component is in execution. In building-block enclave design, developers must carry out security analyses on binaries and utilised third-party packages.

Analysing the TCB size The interpreter enclaves of this analysis have used the trust relationship shown in Fig. 2 (Approach 2). The Interpreter Enclave must, therefore, be public to gain the trust of the *AO*. Unfortunately, TrustJS framework is closed-sourced; i.e., the interpreter enclave is not open for inspection. Thanks to similar open-source framework SecureJS, we were able to analyse its TCB. Table 3 shows the size[14] of the Software TCB components in the SecureJS framework. The MuJS Interpreter ca 13 KLoC takes %92 of the full TCB (ca 14 KLoC). The rest of the interpreter enclave (1137 LoC) includes a small loader, a decrypter code base (excluding the crypto implementations) and one *ECALL* that handles all enclave operations in a single flow. For the components other than the interpreter, a formal verification might be feasible. The MuJS executes the secret Javascript code, and processes the secret parameters of given secret Javascript functions. By porting the MuJS interpreter to SGX, the compiled binary can provide the required functionality, however, the SGX hardware cannot turn an insecure design into a secure one. In fact, an incorrect composition weakens the security guarantees of the hardware. The Interpreter Enclave must satisfy the secrecy requirement for confidential execution. These frameworks rely on implementation of MuJS for confidential execution. As a consequence, using the commodity software package for secret processing becomes a weakness due to the confidentiality requirement.

Table 3 TCB Components and their size in line of C code. Based on software TCB size of SecureJS

TCB component	TCB size (LoC)
Main ECALL	110
Internal enclave functions	387
Crypto functions	254
Attestation mechanism	386
MuJS interpreter	13.022
Trusted Intel libraries	Not included

4.2 Weaknesses in MuJS interpreter

MuJS is a lightweight Javascript interpreter, not designed in the security domain. The *ED* must, therefore, be careful about including third-party source code in the TCB. For this paper, we debugged the MuJS interpreter for the input dependent methods including the data access and the control flow. These particular weaknesses may leak the confidential source code and the confidential parameters. Therefore, our aim is to observe two parts of confidential information on source code; first leaking the Javascript functions, and secondly, leaking the secret string parameters.

Leaking the called Javascript functions via data-dependent control flow weakness MuJS parses Javascript code to extract the necessary operation code (`opcode`). The `opcode` for the JS function is used to call the relevant C method that corresponds with the Javascript method. The corresponding C methods are placed in different memory pages. We report that the control flow in the `jsrun.c` file containing the `jsR_run` method shown in Fig. 5 is dependent on the input parameter of `opcode` containing the Javascript method. The `jsR_run` method is used to parse and call C methods. This may leak the function names of confidential Javascript methods. The control-flow is therefore, non-oblivious.

Leaking the string parameters via data-dependent data access weakness in MuJS MuJS contains `utf8` and `rune` string operations for performing string transformation operations. The string operations are performed over look-up tables. The number of accesses to a look-up table tells the position of the character searched in given table. An often called function for table look-up `bsearch` in `utftype.c` exposed in Fig. 5 may leak the position of the character in the table. This means the execution has non-oblivious data-access.

5 Managing the Software-TCB on SGX enclaves

Both of the fundamental security notions, namely integrity and confidentiality management, require hardware and software co-design and implementation. A secure hardware (a TEE or SGX v1.0 in practice) provides limited security guarantees if its software contains weaknesses. For example, a buffer overflow vulnerability allows an attacker to break the integrity of the target software. If an enclave contains a run-time vulnerability, attackers may take the control of the execution, compromising its integrity. The *ED* should avoid any run-time vulnerability to ensure the integrity guarantees of underlying instructions.

For confidentiality management, SGX can provide page-level secrecy in its enclave memory. The content of a

A

```
//...
opcode =   *pc++;
switch  (opcode) {
case OP_POP:  js_pop(J, 1);  break;
case OP_DUP:  js_dup(J);  break;
case OP_DUP2:  js_dup2(J);  break;
case OP_ROT2:  js_rot2(J);  break;
case OP_ROT3:  js_rot3(J);  break;
case OP_ROT4:  js_rot4(J);  break;

case OP_NUMBER_0: js_pushnumber(J, 0);
break;
case OP_NUMBER_1: js_pushnumber(J, 1);
break;
case OP_NUMBER_POS: js_pushnumber(J,
*pc++); break;
case OP_NUMBER_NEG: js_pushnumber(J,
-(*pc++)); break;
case OP_NUMBER: js_pushnumber(J,
NT(*pc++)); break;
case OP_NUMBER: js_pushliteral(J,
ST(*pc++)); break;
//...
```

B

```
//…
static Rune*
bsearch(Rune c, Rune *t, int n, int ne)
{
  Rune *p;
  int m;

  while(n > 1) {
    m = n/2;
    p = t + m*ne;
    if(c >= p[0]) {
      t = p;
      n = n-m;
    } else
      n = m;
  }
  if(n && c >= t[0])
    return t;
  return 0;
}
//…
```

```
p = bsearch(c, __tolower2, nelem(__tolower2)/3, 3);
```

Fig. 5 Disadvantage of Third-party Packages for Confidentiality Management in Enclaves. Direct port of commodity software may leave additional side-channel traces which ruins the confidentiality guarantees of the hardware. **a**) Excerpt showing the Input Dependent Control Flow in `js_run` method of `jsrun.c` in MuJS interpreter. **b**) Excerpt showing the Input Dependent Data Access in `bsearch` method of `utftype.c` in MuJS interpreter. (Both Accessed on May 2018 Revision.)

memory page is encrypted by the hardware. An enclave that contains commodity software (i.e., unmodified binary programmed without security in mind) may allow attackers to learn the secrets processed by that software. The *ED* should follow secure programming practices.

SGX hardware threat model states the *ED*'s responsibilities in *SGX Blogs* and *SGX Documentations* [25–29] since the time of launching SGX in 2015 (SGX Programming Reference, SGX Developers Guide, SGX Developer Reference, SGX Enclave Writers Guide). The *ED* must keep their development environment malware-free. During programming or compiling an enclave, a malware may infect the enclave and perform malicious operations. The *ED* must also keep their enclave code vulnerability-free. An important post by Intel [29] mentioned that[15] the type of side-channel attack identified on the RSA implementation was well-known. Developers must avoid any weak or vulnerable code in the enclave that plays a role in an attack.

Secure programming techniques can help to keep enclaves free against runtime bugs. The *ED* has the responsibility to program their enclaves with resistance to software-based side-channel attacks. SGX hardware cannot bring automatic security guarantees for an enclave that contains vulnerabilities.

SGX hardware can provide integrity control for memory accesses so that an enclave can only access to its own address space. The untrusted operating system, nevertheless, controls the allocation of the enclave memory pages. By design, SGX cannot provide any protection against any denial of service attacks; if the operating system refuses to give the enclave resources, they cannot operate. If the enclave binary programmed by the *ED* is weak (i.e., performing any data access or control flow based on input data), the operating system can observe the pages requested or used during execution. It has been shown by Hahnel et al. [22] and Tsai et al. [61] how the *ED* can mitigate some potential side-channel attacks.

If the *ED* places vulnerable code inside the SGX enclave, code flaws may cause integrity problems which can be no longer be controlled by SGX (or similar trusted hardware solution). The enclave implementation must contain no code that intentionally causes information leakage through side-channel attacks or covert-channel attacks. These kinds of attacks do not show weakness in SGX technology [29]. The *ED* must analyse its implementation to prevent the side-channel attacks [6, 21, 40] against the software running in SGX enclaves.

Previous studies [10, 18, 47, 49] have shown solutions for page-level and cache-level side-channel attacks. Seo et al. [49] provided a solution for the controlled page-level attacks with a compiler-level scheme, and this [18] study solved the same attack with verifiable page faults. Another study [10], solved the issue of data leakage in side channels via randomisation. Additionally, recent work [47] has enabled Address Space Layout Randomisation (ASLR) for SGX enclaves. The *ED*s may benefit from these solutions. Utilising an oblivious RAM solution [54, 57] may also reduce the attack surface by hiding memory content, and the memory access patterns.

5.1 Bad practices in enclave development

5.1.1 Integrity and confidentiality

SGX technology is a practical TEE solution for securing application secrets. The enclave TCB design is crucial for

providing integrity and confidentiality. For confidentiality management, third-party software included in the TCB may fail to provide secrecy for private algorithms. As such, TCB components that perform sensitive operations require special attention from developers as it is possible for arbitrary software, running on trusted hardware, to leak application secrets. Due to their weak TCB components, TrustJS and SecureJS frameworks may fail to preserve the confidentiality of private algorithms. Secrets must not be processed by arbitrary software unless required security mechanisms are in place.

Impact of TCB size on integrity and confidentiality management Increasing the TCB size reduces the chance of formal verification of the source code. SGX hardware provides strong integrity guarantees for execution and memory isolation. However, a weak software stack included in the TCB may leak secrets that are stored or processed in the enclave. Similarly, a run-time exploit may give control of the enclave to an attacker.

The New Attack Vectors against the software running on trusted hardware Commodity software products and unmodified binaries running on trusted hardware need additional memory protections for confidentiality management. Table 4 shows the *ED*'s responsibilities to preserve security guarantees provided by the trusted hardware. The non-oblivious software stack shown in Section 4 may cause confidentiality leaks, otherwise. Without deploying the security mechanisms that are necessary in a remote environment, the performance metrics becomes obsolete. We list the bad practices in SGX development as following:

- Building a software stack with commodity software: Developers lose security guarantees of the underlying SGX hardware due to weak TCB construction caused by commodity software. Porting the commodity software to SGX does not directly compose it with SGX.

Table 4 Hardware-enhanced security guarantees in enclaves, and the cases when bad software stack may break these guarantees. Enclave Developers (*ED*) are responsible for the secure development

Notion	Hardware feature	May Break
Integrity	Memory access checks	Runtime vulnerability in TCB
Confidentiality	Page-level secrecy	Non-oblivious software stack
Fault tolerance	Sealed storage	Bad software implementation
Enclave availability	Refuses to operate open	Vulnerabilities in microcode

- Runtime vulnerabilities are still a threat: Developers cause serious integrity flaws due to runtime vulnerabilities. Enabling the features of the SGX technology within applications does not mitigate the runtime vulnerabilities. Consequently, a developer who places a runtime vulnerability in the software TCB of a SGX enclave does not show security flaw in the SGX technology.

5.1.2 Two aspects of the availability

Availability of an SGX enclave may refer to two subnotions. First, the SGX enclave binary would refuse to operate if there is no legitimate Intel SGX hardware. This notion is independent of the *ED*. The security issues may arise if SGX instructions have a design flaw. Nonetheless, this would not threaten the validity of enclave research. Because the microcode can receive patches, and the secure hardware solutions in future can avoid the known security issues.

The second point of availability is related to fault tolerance. An enclave-based cloud service must be available. As the operating system controls the resources, it can force an enclave to die. In case of a failure, the enclave can recover from a sealed state. The recovery process would, however, be dependent on how this fault tolerance mechanism is implemented. A bad software implementation may cause enclave to fail on execution, or at recovery. The *ED*, therefore, are responsible for the secure development of an enclave that is resilient to the failures.

6 Secret-code execution (SCE) in reduced TCB

We extended the enclave development model with a new mode called **Early Private Mode** (EPM), shown as **State 1** (S1) in Fig. 7. The EPM helps to SCE in enclaves preserving strong security guarantees and avoids the issues explained in Section 5.1. The overall architecture in Fig. 12 in the Appendix, ① provides confidential execution of private algorithms, ② keeping a minimal TCB size, ③ without the inclusion of any commodity software. The EPM runs when an *AO* executes the enclave in their own execution environment by setting the EPM flag as 'true'. The enclave then outputs the given function reference by reading from runtime memory.

The conventional enclave lifecycle contains three states. The developer firstly compiles the enclave binary, which is then delivered and executed, before attesting the identity of the loaded enclave binary. In our design, we added an Early Private Mode (EPM; or State 1; S1) to serialise the secret code before compiling the standard enclave binary. The S1 serialises the given secret internal enclave function and outputs it to an encrypted blob. After verifying the enclave identity via remote attestation,

we load the serialised secret code for execution (Fig. 8, **State 5**; S5).

6.1 Further enclave partitioning: public and private internal enclave functions

By default, in enclave development [4], binaries are split into trusted and untrusted parts. Trusted enclave part includes a ***Trusted Interface*** (TI), ***ECALLs*** and ***OCALLs***, along with ***Internal Enclave Functions*** (IEF). We extend this paradigm with ***Public IEF*** (PIEF), and ***Secret IEF*** (SIEF) as shown in Fig. 6. In S1 (Fig. 7), the *AO* produces ***Serialised Secret Internal Enclave Function*** (SSIEF) called Asset 1 (A1) within EPM. At this early stage, a copy of the SIEF is created and sealed to enclave state, to be loaded later in S5 (After Remote Attestation) (Fig. 8).

The second state (S2) is not different from the ordinary release mode[16], but includes one additional configuration parameter. By setting the EPM flag as 'false', the build process excludes the Private Enclave Part, and outputs the standard enclave binary. The compiled binary includes the TI, ECALL and OCALL functions, the PIEF and the Private Code (PC) Loader.

6.2 Late-load of secret code at the fifth state

Before execution of the enclave at **State 3** (S3), the *HO* has a chance to investigate the enclave code known as Asset 2 (A2). A2 will contain no secrets that have been embedded in advance. First, the A2 is loaded into the main memory at **State 4** (S4) via the SGX APIs triggered by the ***Untrusted Application*** (UApp) binary. Afterwards, the *HO* cannot see the content of the A2 in the enclave memory (other than the side-channel footprints explained in Section 5). To reduce information leaks via side-channels, we keep the TCB clean from third-party commodity software, not processing any secrets in the PIEF. After Remote Attestation (either TLS-based [32, 55] or Diffie-Hellman & SigMA based [27]) between the AO and A2 (i.e., the AO attests the A2 enclave binary), the PC Loader extracts A1, including the SSIEF, to the pre-allocated executable memory. To execute the secret code, A2 calls the address of A1, passing the execution flow. This operation provides runtime recovery of the secret code invisible to the *HO*.

6.3 Managing security: adversarial AO vs adversarial HO

In addition to the classic threat model of SGX applications, where malicious OS and software stack threaten the enclave, we consider a two-way adversarial case. The *HO* aims to learn the secret code sent by the *AO*. To do so, the *HO* can query the SIEF offline with all possible input set, depending on the application. In order to prevent this, the *AO* has to bind the SIEF to the *HO*'s input parameter, while attesting A2. The parameterisation of the SIEF is kept out of the scope in this paper. Nevertheless, *AO* can mitigate the offline-querying attacks by locking the SIEF into a specific input. In contrast, the *AO* may take control of the enclave via a malicious SSIEF in order to signal back any *HO*-specific private data used in the PIEF or SIEF. This possibility threatens the privacy of the *HO* if it represents multiple end-users (Data Owners, *DO*). As a countermeasure, the *HO* can physically limit any information leakage

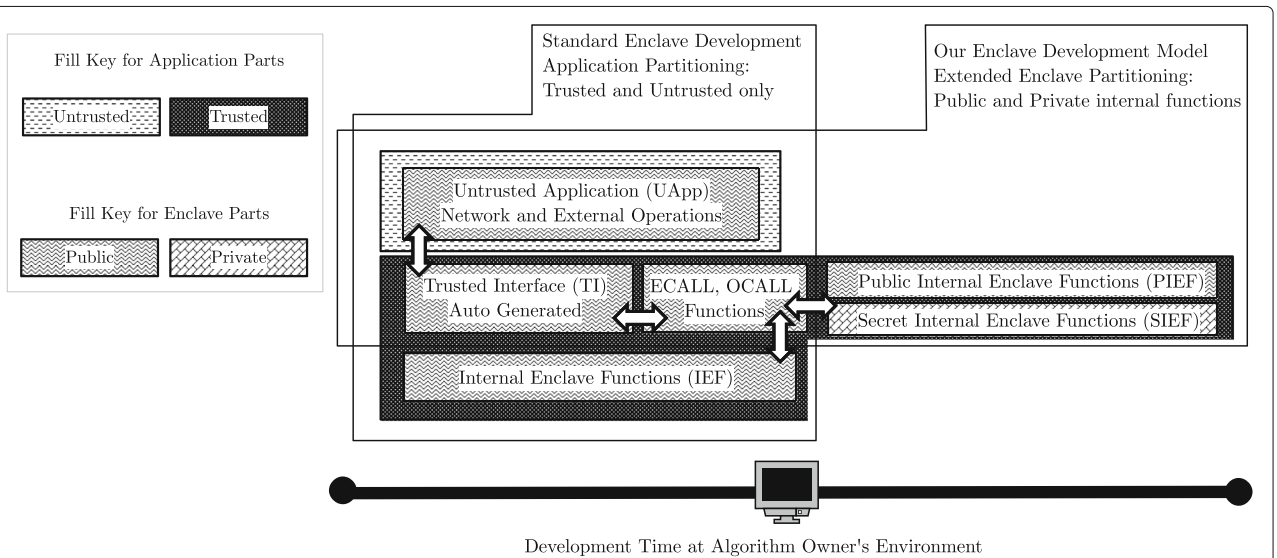

Fig. 6 Extending the Enclave development model with Private and Public parts. In addition to the application partitioning into trusted and untrusted, we introduce the *enclave partitioning* for internal functions. These functions never communicate directly with the outside world

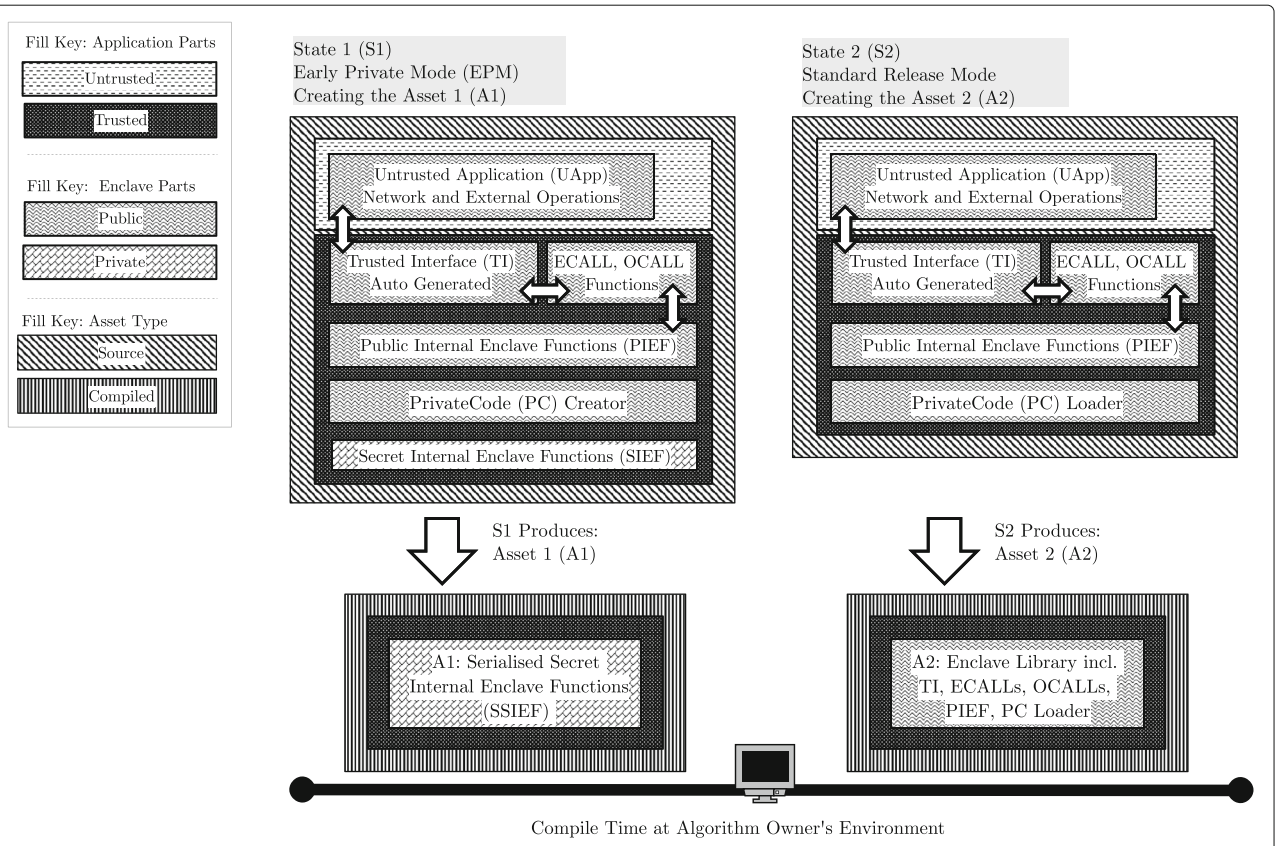

Fig. 7 The Early Private Mode (EPM) runs before releasing the enclave binary. It creates the Asset 1 (A1) that includes the Serialised Secret Internal Enclave Functions (SSIEF). The standard release mode in State 2 (S2) creates the enclave binary including the Private Code Loader

by isolating their environment after loading the SIEF. We do not, however, consider PIEF to be malicious against the *HO*, as it must be open for inspection, and it must be trusted by both parties. This requirement of PIEF being trusted, comes from standard enclave development where the all enclave code is being public. In our model, we keep a part of the enclave private, while rest of it is still known and being open for inspection. In short, the *AO* can lock the SIEF against offline attacks targeting code secrecy, and the *HO* can control the physical execution environment including the network infrastructure to mitigate the covert-channels.

6.4 Comparing the Approach 1 and Approach 2

The secret code execution in reduced TCB (Approach 1) uses direct export and import of the C code. The interpreter enclave method (Approach 2) enables the use of high-level languages.

Usability Approach 1 requires the *AO* to be aware of enclave development, and it requires them to take an additional step of EPM before releasing the code. In Approach 2, the *AO* does not need to have information about the enclave development, as the interpreter provides a layer of abstraction.

Security Using an interpreter leaves more memory traces, potentially leaking information via side-channels. It requires oblivious interpreters and oblivious memory layout in order to hide the control flow. Loading native C functions with the size of memory pages do not leave further traces (at page-level granularity), other than the number of calls.

TCB size Even though using a very small interpreter, Approach 2 dramatically increases the TCB while weakening the threat model. The native C execution method in Approach 1, the TCB contains approx. 1500 LoC for PC Loader, Attestation and other Enclave functions (excluding any application-specific functionalities of the enclave). In comparison to interpreter enclaves of 14 KloC, our method provides a TCB that is at least 10 times smaller. Also, Approach 1 addresses a stronger

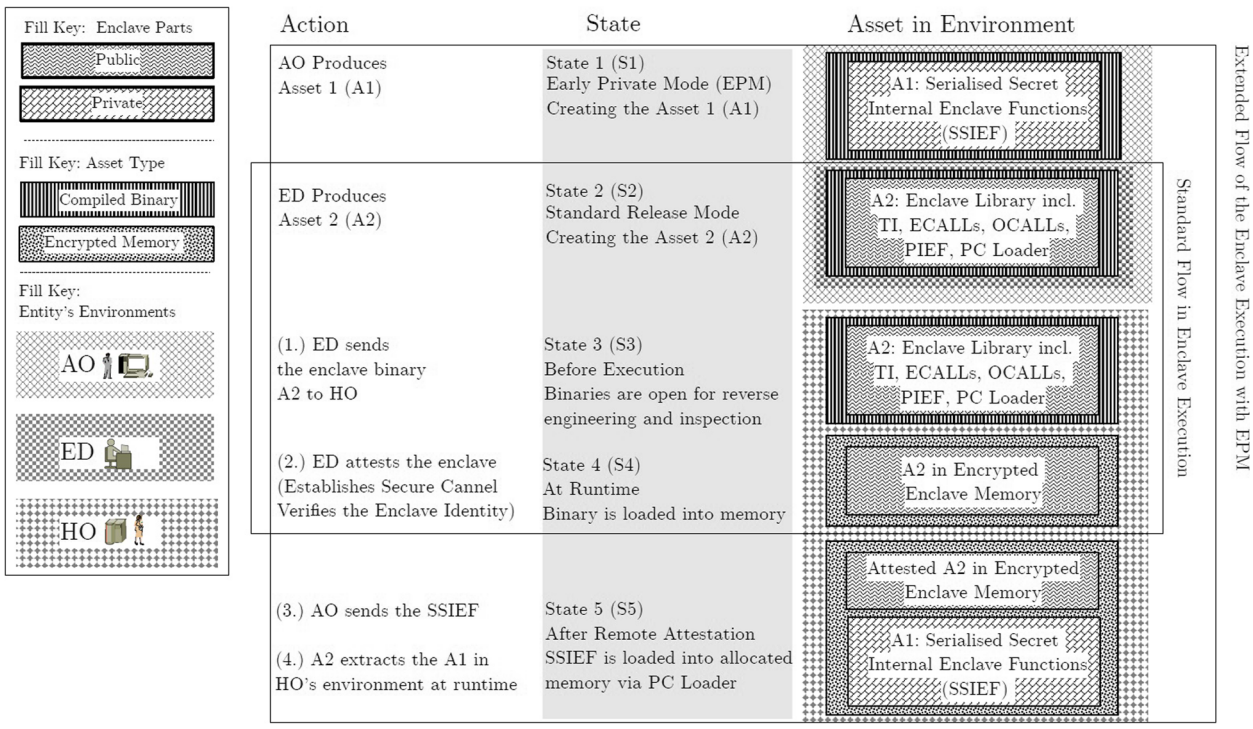

Fig. 8 The *Standard Execution* of an enclave, and the *Extended Execution* flow with EPM for private algorithms. The difference is that the *AO* takes also a role in development of the enclave for private parts. The Asset 1 containing the private algorithms is loaded late in the *HO*'s environment for secret-code execution after verifying the attestation report

threat model towards side-channel attacks by leaving less footprints.

7 Industrial and practical use cases

The *AO* needs to run private algorithms on a remote, untrusted computer. We show (Table 5) three examples where our execution model can achieve this goal. These scenarios differ from each other by the execution environments and the participant who receives the result.

7.1 Secret-code execution on computational power (SCE-CP)

In the first scheme, the *AO* has a set of private algorithms, such as ① a private compression algorithm and ② a private sorting algorithm. These algorithms need high computational power. The *AO*, therefore, rents a remote server from the *HO*. This computation includes embedded constant values and the algorithm provided by the *AO* only. The *DO* does not involve in this scenario. The *AO* wants to get the secret output of the computation.

7.1.1 SCE-CP in practice

The *AO* creates the SSIEF containing the algorithm and the enclave binary. ① The *AO* sends the enclave binary to the *HO*. The *HO* executes the enclave on a SGX-enabled machine. ② The *AO* attests the enclave, and establishes a secure channel. ③ After verifying the enclave identity, the *AO* sends the encrypted SSIEF. The *HO* loads the encrypted content and execution begins. The result of the computation is encrypted with the *AO*'s key. ④ The *AO* receives the result and decrypts it.

Table 5 Three use cases on secret-code execution through SGX enclaves

No.	Use case for	Alg. status	Data status	Result	Execution
1	Computing power (CP)	Private Alg. by AO	Public/no data by DO	A.O. gets	Only at H.O.'s environment
2	Algorithm querying (AQ)	Private Alg. by AO	Private data by DO	D.O. gets	D.O. controls H.O.'s env.
3	Data querying (DQ)	Private Alg. by AO	Private data by DO	A.O. gets	D.O. controls H.O.'s env.

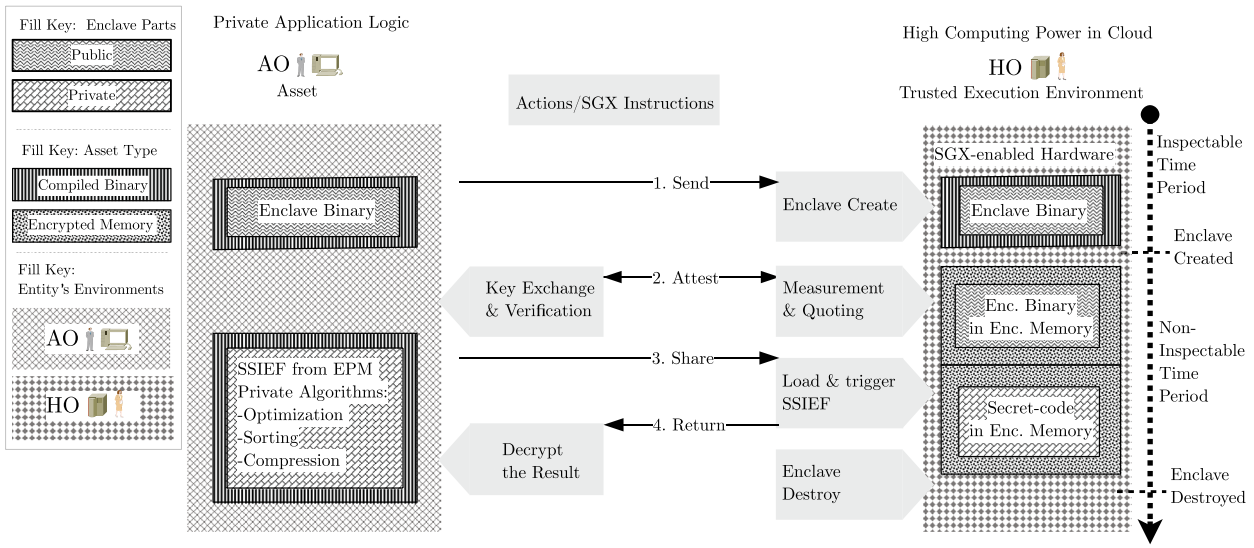

Fig. 9 Secret-Code Execution (SCE) Computational Power. The *AO* creates a session with the enclave binary, and shares the SSIEF. The SSIEF is extracted into the enclave memory at runtime

7.1.2 SCE-CP establishing the mutual trust

The *AO* designs and implements the enclave binary. Sharing the SSIEF after the attestation process gives the *AO* an inherited trust. The SSIEF is decrypted and executed only inside the *Non-Inspectable Time Period* (in Fig. 9). This scheme removes the chance of the *HO* to disassemble any content of the enclave code.

7.1.3 SCE-CP malware in SGX argument

There is a long-lasting argument [46] that an SGX enclave may include malicious code. We evaluate this argument for the SCE-CP use case. Our execution model allows the *AO* to include arbitrary software in the enclave. Without the use of EPM and SSIEF, the *AO* may send the enclave code in plain text, or the *AO* may entirely avoid utilising enclaves. This, however, does not stop the *AO* from sending malware to the cloud environment. In fact, the *AO* is free to run an experiment with malicious or benign code in the cloud. At the bottom, the *HO* controls the hardware resources, and observes the usages. The *HO* charges more to the *AO*, if the resources are used more. At all times, the *HO* can observe all I/O traffic of the enclave. The *HO* can refuse to give resource at any time. The enclave, resource-wise, is one of the most visible parts in the system. Through the enclave or not, the malicious *AO* has full access to the cloud machine. We conclude that use of an enclave does not increase the existing attack surface in the SCE-CP scenario.

7.2 Secret-code execution on algorithm querying (SCE-AQ)

In the second scheme, the *DO* has private input data, such as ① an image containing health data and ② a data-set collected from sensors. As an untrusted entity, the *AO* has a private algorithm that can process this private input. The SCE-AQ scheme (Fig. 10) outputs a private value that the *DO* must receive only. We assume that both entities are mutually distrustful. They do not want to share their private assets with each other. A collusion between the *AO* and the *HO* against the *DO* can leak the secret data. We, therefore, consider the *HO* to be controlled by the *DO*.

7.2.1 SCE-AQ in practice

The *AO* uses the EPM and creates the SSIEF containing the algorithm, and the enclave binary. ① The *AO* sends the enclave binary to the *HO* that is controlled by the *DO*. ② Different from the SCE-CP, the *DO* interacts first with the enclave and commits the input. ③ The *AO* attests the enclave and verifies the identity. ④ If convinced, the *AO* shares the SSIEF with the enclave. The key point is that neither the *DO* nor the enclave can change the execution after this stage. ⑤ The result of the computation is returned to the *DO*.

7.2.2 SCE-AQ brute-forcing the algorithm secrecy

This scheme returns the computation result to the *DO*. This allows the *DO* to gradually learn about the algorithm. A malicious the *DO* may want to find the all possible

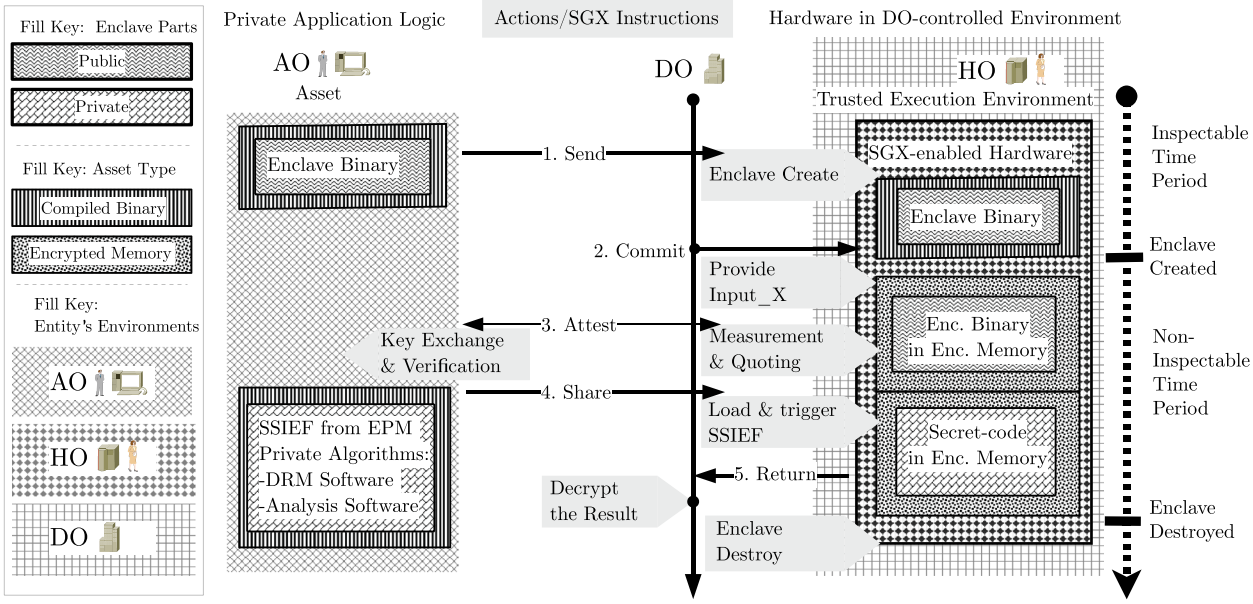

Fig. 10 Secret-Code Execution (SCE) Querying the Algorithm. The *DO* provides an input to the secret algorithm and receives the output. Input commitment prevents querying the secret-code with a different parameter in an offline repeated execution

input and output streams of the algorithm. The Table 6 shows an example to learn the function behaviour by brute forcing the input streams. The *DO* can freeze and clone the memory state of the machine to perform an attack against the algorithm. In our mechanism, however, the independent input from the *DO* comes before the SSIEF. Once the *DO* commits to an input parameter (② step, Fig. 10), the enclave does not accept any other interaction other than accepting the SSIEF (④ step, Fig. 10). If the *DO* clones the enclave at any stage, the result does not change. The *AO* can thus measure how much secrecy of the algorithm has been already leaked.

7.2.3 SCE-AQ preventing the data leaks

In contrast, the *AO* may want to leak the secret inputs. Even though the *AO* can execute any code, the SSIEF cannot communicate with any other entity. The *DO* physically controls the environment, and does not allow any I/O operation (Fig. 11).

Table 6 Brute-force querying the algorithm by re-using the enclave

No.	Example Input_x	Example method	Example Result_y
001/256	00-1111-00	SSIEF(function_f())	11
002/256	11-0101-11	SSIEF(function_f())	10
..	.. ..	SSIEF(function_f())	..
256/256	00-0110-01	SSIEF(function_f())	00

7.3 Secret-code execution on data querying (SCE-DQ)

In the third scenario, the *AO* and the *DO* goes into another joint computation. The difference is that the result must be returned to the *AO* only. The *DO* has a special private data-set. The *AO* wants to run a secret query on this data-set.

Suppose that, the *DO* is an authority who collects road-data and driving experiences of citizens through sensors. In another example, the *DO* can be a car manufacturer company who collects data from their cars. The *AO* is an insurance company, or a government authority, who wants to run a private algorithm on that data-set.

7.3.1 SCE-DQ in practice

Similar to SCE-AQ, ① the *AO* sends the binary, ② the *DO* commits the input, ③ the *AO* attests the enclave, and ④ the *AO* shares the SSIEF. Differently, at step ⑤, the *DO* reduces the bandwidth of the result. ⑥ The *AO* receives the result and decrypts it.

7.3.2 SCE-DQ limiting the data leakage

The difference in the SCE-DQ from the SCE-AQ is that the *AO* receives the secret result. At the end of the computation, the *DO* does not let SSIEF to return an arbitrary value directly. In this example, we allow returning only 1 bit of data (boolean) as a result. This operation means, the *DO* does not know what software has processed her input, and does not know the result.

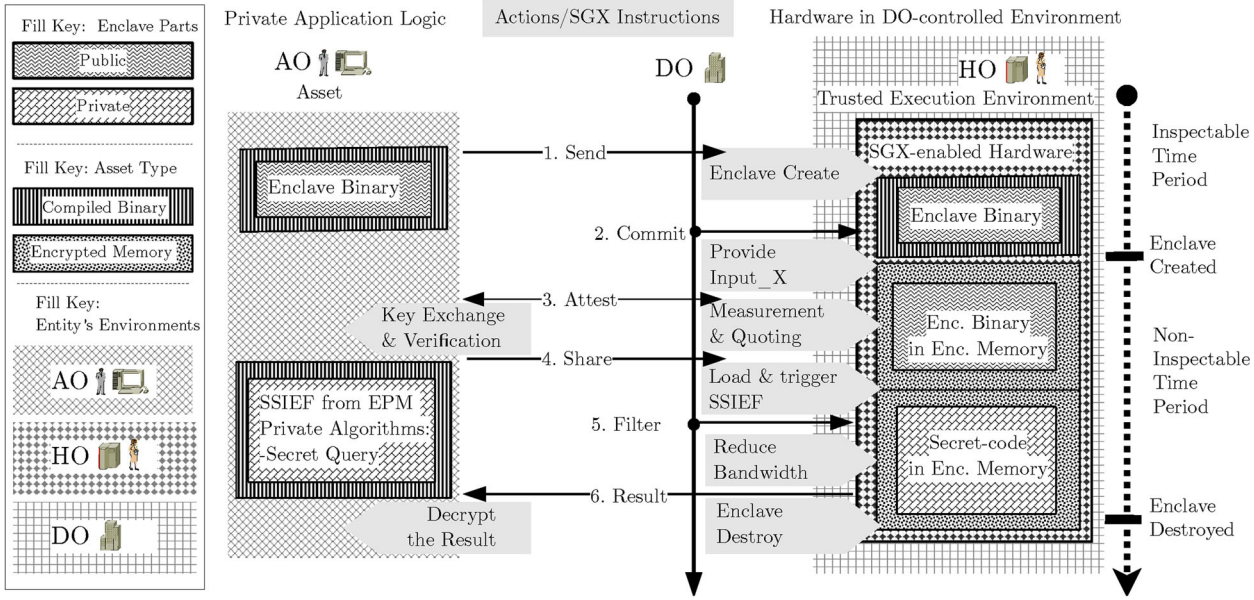

Fig. 11 Secret-Code Execution (SCE) Querying the Data. The *AO* runs a secret query on a secret data set. After the secret-code execution, the *DO* has control of the bandwidth for filtering the return value

The *DO*, however, knows the type of the value, and permits SSIEF to return this secret value to the *AO*. By doing so, the *DO* can control how much secrecy of data is disclosed.

7.4 The overhead in SCE components

We observed two types of computational costs in our use cases: ① The asynchronous operations that are independent of the main computation. ② The synchronous time cost that our model adds on top of the existing cost. Table 7 summarises the memory space (m) and computing time (t) overheads of the components. In the worst case, the SSIEF recovery costs an equal amount of time to the cost of the enclave creation (t). This overhead sums up to Worst($2t$) of total time for the SSIEF execution. The computation itself, however, can take time and memory as required by the *AO*. The SSIEF is executed in the preallocated memory area during enclave creation. This operation, therefore, does not require additional memory. We keep the remote attestation costs and operations out of the scope in this study.

Table 7 Overhead of three use cases on *secret-code execution*

Description	SCE CP	SCE AQ	SCE DQ
Development time public/private partitioning: asynchronous			
Compile time EPM and enclave: asynchronous			
t Time cost, m memory size	1. Step: Execute the enclave		
Worst case $2t$ total time cost	3. Step: Exec SSIEF	4. Step: Exec SSIEF	
Other operations: existing costs			

8 Conclusion and future work

In summary, this paper presents a security analysis on interpreter enclaves and has aimed to demonstrate how a third-party commodity software can become the weakest link in a security chain. We provided a new design for secret-code execution in remote computers, and demonstrated such design in three practical generic templates. Our model reduced the TCB size by a power of ten, in comparison to the alternative design approach using interpreters. Approach 1 brings an extra step in development, requiring additional enclave partitioning (public and private parts). In Approach 2, we showed it is difficult to hide the called functions, thus requiring an additional strong sandbox for function secrecy. This sandbox would, ideally, need to call a set of similar functions for each function. It is very likely to have an exponential computational overhead, and this overhead may grow with the size of the interpreter. Despite its shortcomings on usability, our approach provides stronger security guarantees. Future work may explore automated ways of deploying our design, however, Approach 2 will continue to have better usability. In conclusion, our late-load method for secret-code execution provides stronger security and native execution performance, requiring only a small additional development effort from the Algorithm Owner.

Endnotes

[1] Digital Rights Management in Trusted Computing. https://en.wikipedia.org/wiki/Trusted_Computing#Digital_rights_management visited on 04/Jun/2019.

[2] Security in Golem Network https://docs.golem.network/#/About/Security visited on 04/Jun/2019.

[3] https://www.serecaproject.eu/index.php/publications/papers visited on 04/Jun/2019.

[4] Linux Kernel Library. https://github.com/lsds/sgx-lkl visited on 04/Jun/2019.

[5] https://www.securecloudproject.eu/papers/ visited on 04/Jun/2019.

[6] https://www.securecloudproject.eu/project-overview-securecloud/ visited on 04/Jun/2019.

[7] https://software.intel.com/en-us/sgx/academic-research visited on 04/Jun/2019.

[8] https://github.com/Microsoft/openenclave visited on 04/Jun/2019.

[9] Linux Kernel Library. https://github.com/lsds/sgx-lkl visited on 04/Jun/2019.

[10] http://git.ghostscript.com/?p=mujs.git;a=summary Online Repository. Visited on 04/Jun/2019.

[11] https://github.com/oscarlab/graphene/tree/master/LibOS/shim/test/apps/python visited on 04/Jun/2019.

[12] TrustJS published on 23rd April 2017, the source-code was not published as of writing this paper, on 28th of May 2018.

[13] SecureJS. https://github.com/AsierRF/SecureJS visited on 04/Jun/2019.

[14] Using SLOCCount. https://www.dwheeler.com/sloccount/ visited on 04/Jun/2019.

[15] A recent blog post by Intel [29] stated following: *In general, these research papers do not demonstrate anything new or unexpected about the Intel SGX architecture. Preventing side channel attacks is a matter for the enclave developer.*

[16] Release Mode: https://software.intel.com/en-us/documentation/intel-sgx-web-based-training/debugging-enclaves visited on 04/Jun/2019.

Appendix

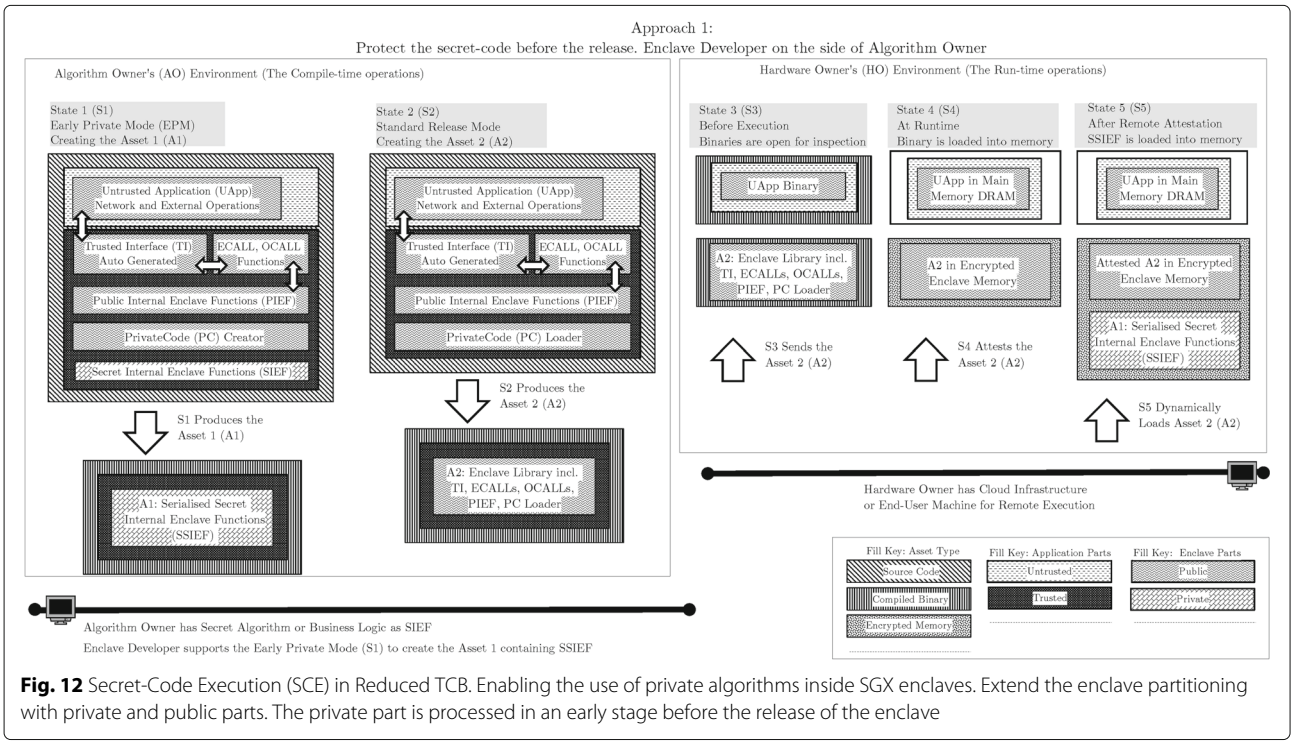

Fig. 12 Secret-Code Execution (SCE) in Reduced TCB. Enabling the use of private algorithms inside SGX enclaves. Extend the enclave partitioning with private and public parts. The private part is processed in an early stage before the release of the enclave

Acknowledgements
We thank Sean Smith, Il Ceylan, M Geden, K Kalkan, E Ucan, IM Tas, and B Sari for their helpful discussions and reviews. We would also like to thank SE Bazley, OJ Sturrock, Y Ulaş, and L Bihr for their reviews.

Funding
None.

Availability of data and materials
Data sharing is not applicable to this article as no datasets were generated or analysed during the current study.

Authors' contributions
KAK contributed to design and implementation of the research, to analysis of results, and writing of the manuscript. DG contributed to the conception and design of this work, and contributed to the production of this manuscript. AM provided supervision and consultation, and contributed to the production of the manuscript. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]Robert Hooke Building, Cyber Security Centre, Computer Science Department, University of Oxford, Oxford, UK. [2]www.TwentySeven34.com.

Received: 21 October 2018 Accepted: 3 May 2019
Published online: 05 September 2019

References

1. Z.A. Al-Sharif, M.I. Al-Saleh, L.M. Alawneh, Y.I. Jararweh, B. Gupta, Live forensics of software attacks on cyber physical systems. Futur. Gener. Comput. Syst (2018). https://doi.org/10.1016/j.future.2018.07.028, visited on 02/Jun/2019
2. I. Anati, S. Gueron, S. Johnson, V. Scarlata, in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. HASP '13*. Innovative Technology for CPU Based Attestation and Sealing, vol. 13 (ACM, New York, 2013), pp. 1–7
3. S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, M.L. Stillwell, D. Goltzsche, D. Eyers, P. Pietzuch, C. Fetzer, in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. OSDI'16*. SCONE: Secure Linux Containers with Intel SGX (USENIX Association, Berkeley, CA, USA, 2016), pp. 689–703. http://doi.acm.org/10.1145/3026877.3026930, visited on 02/Jun/2019
4. A. Atamli-Reineh, A. Martin, in *Security and Privacy in Communication Networks*, ed. by B. Thuraisingham, X. Wang, and V. Yegneswaran. Securing Application with Software Partitioning: A Case Study Using SGX (Springer International Publishing, Cham, 2015), pp. 605–621
5. P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, P. Pietzuch, in *Proceedings of the Thirteenth EuroSys Conference. EuroSys '18*. LibSEAL: Revealing Service Integrity Violations Using Trusted Execution (ACM, New York, 2018), pp. 24:1–24:15. https://doi.org/10.1145/3190508.3190547, visited on 02/Jun/2019
6. F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, A.-R. Sadeghi, in *Proceedings of the 11th USENIX Conference on Offensive Technologies. WOOT'17*. Software Grand Exposure: SGX Cache Attacks Are Practical (USENIX Association, Berkeley, 2017), p. 11. https://doi.acm.org/10.1145/3154768.3154779, visited on 02/Jun/2019
7. S. Brenner, D. Goltzsche, R. Kapitza, in *Proceedings of the 1st International Workshop on Security and Dependability of Multi-Domain Infrastructures. XDOMO'17*. TrApps: Secure Compartments in the Evil Cloud (ACM, New York, 2017), pp. 5:1–5:6. https://doi.acm.org/10.1145/3071064.3071069, https://doi.org/10.1145/3071064.3071069, visited on 02/Jun/2019
8. S. Brenner, T. Hundt, G. Mazzeo, R. Kapitza, in *Distributed Applications and Interoperable Systems*, ed. by L.Y. Chen, H.P. Reiser. Secure Cloud Micro Services Using Intel SGX (Springer International Publishing, Cham, 2017), pp. 177–191
9. F. Campanile, L. Coppolino, S. DAntonio, L. Lev, G. Mazzeo, L. Romano, L. Sgaglione, F. Tessitore, in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. Cloudifying Critical Applications: A Use Case from the Power Grid Domain, (2017), pp. 363–370. https://doi.org/10.1109/PDP.2017.50, visited on 02/Jun/2019
10. S. Chandra, V. Karande, Z. Lin, L. Khan, M. Kantarcioglu, B. Thuraisingham, in *Computer Security – ESORICS 2017*, ed. by S.N. Foley, D. Gollmann, and E. Snekkenes. Securing Data Analytics on SGX with Randomization (Springer International Publishing, Cham, 2017), pp. 352–369. https://doi.org/10.1007/978-3-319-66402-6_21, visited on 02/Jun/2019
11. S. Checkoway, H. Shacham, in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '13*. Iago attacks (ACM Press, New York, p. 253
12. F. Chen, M. Dow, S. Ding, Y. Lu, X. Jiang, H. Tang, S. Wang, PREMIX: PRivacy-preserving EstiMation of Individual admiXture. AMIA Annu. Symp, 1747–1755 (2016). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5333197/pdf/2500255.pdf, visited on 02/Jun/2019
13. F. Chen, S. Wang, X. Jiang, S. Ding, Y. Lu, J. Kim, S.C. Sahinalp, C. Shimizu, J.C. Burns, V.J. Wright, E. Png, M.L. Hibberd, D.D. Lloyd, H. Yang, A. Telenti, C.S. Bloss, D. Fox, K. Lauter, L. Ohno-Machado, PRINCESS: Privacy-protecting Rare disease International Network Collaboration via Encryption through Software guard extensionS. Bioinformatics (Oxford, England). **33**(6), 871–878 (2017). https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5860394/pdf/btw758.pdf, visited on 02/Jun/2019
14. L. Coppolino, S. D'Antonio, G. Mazzeo, G. Papale, L. Sgaglione, F. Campanile, in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. An Approach for Securing Critical Applications in Untrusted Clouds, (2018), pp. 436-440. https://doi.org/10.1109/PDP2018.2018.00076, visited on 02/Jun/2019
15. V. Costan, I. Lebedev, S. Devadas, in *Proceedings of the 25th USENIX Conference on Security Symposium. SEC'16*. Sanctum: Minimal Hardware Extensions for Strong Software Isolation (USENIX Association, Berkeley, 2016), pp. 857–874. https://doi.acm.org/10.1145/3241094.3241161, visited on 02/Jun/2019
16. C. Cremers, Compositionality of Security Protocols: A Research Agenda. Electron. Notes Theor. Comput. Sci. **142**, 99–110 (2006). https://doi.org/10.1016/j.entcs.2004.12.047, visited on 02/Jun/2019. Proceedings of the First International Workshop on Views on Designing Complex Architectures (VODCA 2004)
17. A.R. Fernandez, in *Master's thesis in Computer Systems and Networks*. Integrity and confidentiality for web application code execution in untrusted clients. Promoting a Trust Relation in Web-Applications, (Göteborg, 2017). http://publications.lib.chalmers.se/records/fulltext/252354/252354.pdf, visited on 03/Jun/2019
18. Y. Fu, E. Bauman, R. Quinonez, Z. Lin, in *Research in Attacks, Intrusions, and Defenses*. SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults (Springer International Publishing, Cham, 2017), pp. 357–380
19. Global Platform, *Introduction to Trusted Execution Environments*. (Global Platform, Inc., Non-Profit Association, 2018). Online Resource In Global Platform, Inc. Website. https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf, visited on 03/Jun/2019
20. D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, R. Kapitza, in *Proceedings of the 10th European Workshop on Systems Security. EuroSec'17*. TrustJS: Trusted Client-side Execution of JavaScript (ACM, New York, 2017), pp. 7:1–7:6. https://doi.org/10.1145/3065913.3065917, visited on 03/Jun/2019
21. J. Götzfried, M. Eckert, S. Schinzel, T. Müller, in *Proceedings of the 10th European Workshop on Systems Security. EuroSec'17*. Cache Attacks on Intel SGX (ACM, New York, 2017), pp. 2:1–2:6. https://doi.org/10.1145/3065913.3065915, visited on 03/Jun/2019
22. M. Hähnel, W. Cui, M. Peinado, in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference. USENIX ATC '17*. High-resolution Side Channels for Untrusted Operating Systems (USENIX Association, Berkeley, 2017), pp. 299–312. https://doi.acm.org/10.1145/3154690.3154719, visited on 03/Jun/2019
23. M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, J. Del Cuvillo, in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy. HASP '13*. Using Innovative Instructions to Create Trustworthy Software Solutions (ACM, New York, 2013), pp. 11:1–11:1. https://doi.org/10.1145/2487726.2488370, visited on 03/Jun/2019

24. T. Hunt, Z. Zhu, Y. Xu, S. Peter, E. Witchel, in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. OSDI'16*. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data (USENIX Association, Berkeley, 2016), pp. 533–549. https://doi.acm.org/10.1145/3026919, visited on 03/Jun/2019
25. Intel, *Intel® Software Guard Extensions Programming Reference*, Ref. #329298-002. (Intel Corporation, Portland, 2014). https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, visited on 03/Jun/2019
26. Intel, *Intel® Software Guard Extensions Enclave Writer's Guide v1.02*, Revision 1.02. (Intel Corporation, Portland, 2015). https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf, visited on 03/Jun/2019
27. Intel, *Intel ® Software Guard Extensions Developer Reference (Intel ® SGX) SDK for Linux * OS*, Revision 2.1. (Intel Corporation, Portland, 2017). https://download.01.org/intel-sgx/linux-2.1/docs/Intel_SGX_Developer_Reference_Linux_2.1_Open_Source.pdf, visited on 03/Jun/2019
28. Intel, *Intel® Software Guard Extensions (Intel® SGX) Developer Guide v2.1*, Revision 2.1 Linux. (Intel Corporation, Portland, 2019). https://download.01.org/intel-sgx/linux-2.1/docs/Intel_SGX_Developer_Guide.pdf, visited on 03/Jun/2019
29. Intel, SJ, *Intel® SGX and Side-Channels*. (Intel Corporation, Portland. Developer Zone, https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, published on March 16th 2017, updated February 27th 2018, visited on 03/Jun/2019
30. K. John, M. Roger, 2016 Data Science Salary Survey (2016). https://www.oreilly.com/data/free/2016-data-science-salary-survey.csp, visited on 03/Jun/2019
31. F. Kelbert, F. Gregor, R. Pires, S. Kpsell, M. Pasin, A. Havet, V. Schiavoni, P. Felber, C. Fetzer, P. Pietzuch, in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. SecureCloud: Secure big data processing in untrusted clouds, (2017), pp. 282-285. https:/doi.org/10.23919/DATE.2017.7926999, visited on 03/Jun/2019
32. T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, M. Vij, *Integrating Remote Attestation with Transport Layer Security*. (Intel Corporation, Portland, 2018). https://arxiv.org/abs/1801.05863, visited on 03/Jun/2019
33. R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, C. Fetzer, in *Proceedings of the Thirteenth EuroSys Conference. EuroSys '18*. Pesos: Policy Enhanced Secure Object Store (ACM, New York, 2018), pp. 25:1–25:17. https://doi.org/10.1145/3190508.3190518, visited on 03/Jun/2019
34. K.A. Küçük, A. Paverd, A. Martin, N. Asokan, A. Simpson, R. Ankele, in *Proceedings of the 1st Workshop on System Software for Trusted Execution. SysTEX '16*. Exploring the Use of Intel SGX for Secure Many-Party Applications (ACM, New York, 2016), pp. 5:1–5:6. https:/doi.org/10.1145/3007788.3007793, visited on 03/Jun/2019
35. D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, C. Fetzer, in *Proceedings of the Twelfth European Conference on Computer Systems. EuroSys '17*. SGXBOUNDS: Memory Safety for Shielded Execution (ACM, New York, 2017), pp. 205–221. https://doi.org/10.1145/3064176.3064192, visited on 03/Jun/2019
36. J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, P. Pietzuch, in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference. USENIX ATC '17*. Glamdring: Automatic Application Partitioning for Intel SGX (USENIX Association, Berkeley, 2017), pp. 285–298. https://doi.acm.org/10.1145/3154690.3154718, visited on 03/Jun/2019
37. C. Linn, S. Debray, in *Proceedings of the 10th ACM Conference on Computer and Communications Security. CCS '03*. Obfuscation of Executable Code to Improve Resistance to Static Disassembly (ACM, New York, 2003), pp. 290–299. https://doi.org/10.1145/948109.948149, visited on 03/Jun/2019
38. J.M. McCune, B. Parno, A. Perrig, M.K. Reiter, A. Seshadri, in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XIII*. How Low Can You Go?: Recommendations for Hardware-supported Minimal TCB Code Execution (ACM, New York, 2008), pp. 14–25. https://doi.org/10.1145/1346281.1346285, visited on 03/Jun/2019
39. F. McKeen, I. Alexandrovich, A. Berenzon, C.V. Rozas, H. Shafi, V. Shanbhogue, U.R. Savagaonkar, in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy. HASP '13*. Innovative Instructions and Software Model for Isolated Execution (ACM, New York, 2013), pp. 10:1–10:1. https://doi.org/10.1145/2487726.2488368, visited on 03/Jun/2019
40. A. Moghimi, G. Irazoqui, T. Eisenbarth, in *Cryptographic Hardware and Embedded Systems – CHES 2017*, ed. by W. Fischer, N. Homma. CacheZoom: How SGX Amplifies the Power of Cache Attacks (Springer International Publishing, Cham, 2017), pp. 69–90
41. O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, M. Costa, in *Proceedings of the 25th USENIX Conference on Security Symposium. SEC'16*. Oblivious Multi-party Machine Learning on Trusted Processors (USENIX Association, Berkeley, 2016), pp. 619–636. https://doi.acm.org/10.1145/3241094.3241143, visited on 03/Jun/2019
42. F. Piessens, D. Devriese, J.T. Mhlberg, R. Strackx, in *2016 IEEE Cybersecurity Development (SecDev)*. Security Guarantees for the Execution Infrastructure of Software Applications, (2016), pp. 81-87. https://doi.org/10.1109/SecDev.2016.030, visited on 03/Jun/2019
43. D.RK. Ports, T. Garfinkel, in *Proceedings of the 3rd Conference on Hot Topics in Security. HOTSEC'08*. Towards Application Security on Untrusted Operating Systems (USENIX Association, Berkeley, 2008), pp. 1:1–1:7. https://doi.acm.org/10.1145/1496671.1496672
44. R.J. Riella, L.M. Iantorno, L.CR. Junior, D. Seidel, K.VO. Fonseca, L. Gomes-Jr, M.O. Rosa, in *Proceedings of the 1st Workshop on Privacy by Design in Distributed Systems. W-P2DS'18*. Securing Smart Metering Applications in Untrusted Clouds with the SecureCloud Platform (ACM, New York, 2018), pp. 5:1–5:6. https://doi.org/10.1145/3195258.3195263, visited on 03/Jun/2019
45. F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, M. Russinovich, in *Proceedings of the 2015 IEEE Symposium on Security and Privacy. SP '15*. VC3: Trustworthy Data Analytics in the Cloud Using SGX, vol. 2015-July (IEEE Computer Society, Washington, DC, 2015), pp. 38–54. https://doi.org/10.1109/SP.2015.10, visited on 03/Jun/2019
46. M. Schwarz, S. Weiser, D. Gruss, Practical Enclave Malware with Intel SGX. arXiv preprint arXiv:1902.03256 (2019). http://arxiv.org/abs/1902.03256, visited on 03/Jun/2019
47. J. Seo, B. Lee, S.M. Kim, M.-W. Shih, I. Shin, D. Han, T. Kim, in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs (The Internet Society, Reston, VA, 2017). http://dblp.org/rec/bibtex/conf/ndss/SeoLKSSHK17, visited on 03/Jun/2019
48. F. Shaon, M. Kantarcioglu, Z. Lin, L. Khan, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. CCS '17*. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors (ACM, New York, 2017), pp. 1211–1228. https://doi.org/10.1145/3133956.3134095, visited on 03/Jun/2019
49. M.-W. Shih, S. Lee, T. Kim, M. Peinado, in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs (The Internet Society, Reston. https://dblp.org/rec/bib/conf/ndss/Shih0KP17, visited on 03/Jun/2019
50. S. Shinde, D.L. Tien, S. Tople, P. Saxena, in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. PANOPLY: Low-TCB Linux Applications with SGX Enclaves The Internet Society, Reston, 2017). https://dblp.org/rec/bib/conf/ndss/ShindeTTS17, visited on 03/Jun/2019
51. L.V. Silva, R. Marinho, J.L. Vivas, A. Brito, in *Proceedings of the Symposium on Applied Computing. SAC '17*. Security and Privacy Preserving Data Aggregation in Cloud Computing (ACM, New York, 2017), pp. 1732–1738. https://doi.org/10.1145/3019612.3019795, visited on 03/Jun/2019
52. L. Singaravelu, C. Pu, H. Härtig, C. Helmuth, in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. Reducing TCB complexity for security-sensitive applications, vol. 40 (ACM, New York, 2006), pp. 161–174. https://doi.org/10.1145/1217935.1217951, visited on 03/Jun/2019
53. R. Sinha, S. Rajamani, S. Seshia, K. Vaswani, in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. Moat: Verifying Confidentiality of Enclave Programs (ACM Press, New York, 2015), pp. 1169–1184. https://doi.org/10.1145/2810103.2813608, visited on 03/Jun/2019
54. E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, S. Devadas, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. CCS '13*. Path ORAM: An Extremely Simple

Oblivious RAM Protocol (ACM, New York, 2013), pp. 299–310. https://doi.org/10.1145/2508859.2516660, visited on 03/Jun/2019
55. M. Steiner, T. Knauth, L. Lei, B. Xing, M. Vij, S. Chakrabarti, Technology For Establishing Trust During A Transport Layer Security Handshake (2019). https://patents.google.com/patent/US20190065406A1/en, visited on 03/Jun/2019. US Patent App. 16/174,337. Intel Corporation. In Google Patents
56. R. Strackx, F. Piessens, in *Proceedings of the 1st Workshop on System Software for Trusted Execution - SysTEX '16*. Developing Secure SGX Enclaves: New Challenges on the Horizon (ACM Press, New York, 2016), pp. 3:1–3:2. https://doi.org/10.1145/3007788.3007791, visited on 03/Jun/2019
57. T.P. Thao, A. Miyaji, M.S. Rahman, S. Kiyomoto, A. Kubota, in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*. Robust ORAM: Enhancing Availability, Confidentiality and Integrity (IEEE, Christchurch, 2017), pp. 30-39. https://doi.org/10.1109/PRDC.2017.14, visited on 03/Jun/2019
58. C.-C. Tsai, D.E. Porter, M. Vij, in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference. USENIX ATC '17*. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX (USENIX Association, Berkeley, 2017), pp. 645–658. https://doi.acm.org/10.1145/3154690.3154752, visited on 03/Jun/2019
59. N. Weichbrodt, A. Kurmus, P. Pietzuch, R. Kapitza, in *21st European Symposium on Research in Computer Science, Computer Security – ESORICS 2016*. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves (Springer International Publishing, Cham, 2016), pp. 440–457. https://doi.org/10.1007/978-3-319-45744-4_22, visited on 03/Jun/2019
60. Z. Wu, S. Gianvecchio, M. Xie, H. Wang, in *Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS '10*. Mimimorphism: A New Approach to Binary Code Obfuscation (ACM, New York, 2010), pp. 536–546. https://doi.org/10.1145/1866307.1866368, visited on 03/Jun/2019
61. Y. Xu, W. Cui, M. Peinado, in *Proceedings of the 2015 IEEE Symposium on Security and Privacy. SP '15*. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems (IEEE Computer Society, Washington, DC, 2015), pp. 640–656. https://doi.org/10.1109/SP.2015.45, visited on 03/Jun/2019