

Enclave Management Models for Safe Execution of Software Components

Newton Carlos Will¹^a and Carlos Alberto Maziero²^b

¹Computer Science Department, Federal University of Technology, Paraná, Dois Vizinhos, Brazil

²Computer Science Department, Federal University of Paraná, Curitiba, Brazil

Keywords: Intel SGX, Programming Models, Software Architecture, Performance, Resource Optimization.

Abstract: Data confidentiality is becoming increasingly important to computer users, both in corporate and personal environments. In this sense, there are several solutions proposed to maintain the confidentiality and integrity of such data, among them the Intel *Software Guard Extensions (SGX)* architecture. The use of such mechanisms to provide confidentiality and integrity for sensitive data imposes a performance cost on the application execution, due to the restrictions and checks imposed by the Intel SGX architecture. Thus, the efficient use of SGX enclaves requires some management. The present work presents two management models for using SGX enclaves: (i) enclave sharing; and (ii) enclave pool. In order to apply such models, an enclave provider architecture is proposed, offering a decoupling between the enclave and the application, allowing to apply the proposed management models and offering the resources provided by the enclaves to the applications through an “as a service” approach. A prototype was built to evaluate the proposed architecture and management models; the experiments demonstrated a considerable reduction in the performance impact for enclave allocation, while guaranteeing good response times to satisfy simultaneous requests.

1 INTRODUCTION


Technology is present in the lives of most people, who trust their data to devices such as computers, smartphones, and online storage services. Data present on such devices are quite diverse, from personal photos, passwords, calendars, medical and banking data, and work-related information, which can be very important to users. It is thus necessary to specify mechanisms and practices to ensure their confidentiality and integrity.


Several mechanisms have been developed in order to support operating systems, libraries, and applications that aim to ensure trusted execution and handling of sensitive user data. One of the most recent technologies in this area is the Intel *Software Guard Extensions (SGX)*, which allows an application to be split into two components: a trusted (called *enclave*) and an untrusted one. Intel SGX brings a series of new mechanisms and instructions that aim to ensure the confidentiality and integrity of the data being manipulated inside each enclave, which holds the trusted

component of the application (McKeen et al., 2013).

SGX security guarantees add considerable performance overhead to both creating enclaves and running code within them. Furthermore, the memory region used by SGX is limited, restricting the size and number of active enclaves (Shaon et al., 2017; Mofrad et al., 2017; Fuhry et al., 2017). An issue raised is the possibility of reducing the performance impact inherent to the use of enclaves for handling sensitive data, especially at their creation time (Fisch et al., 2017). Another issue to be tackled is the possibility of reducing the use of the SGX trusted memory, allowing more enclaves to be allocated simultaneously. Such questions are very important for secure applications that should satisfy several requests in a short period of time, considering that the initialization of an enclave to satisfy each request would be costly, from the point of view of performance and memory usage.

Some works in the literature aim to improve the performance of applications that use the Intel SGX architecture, allowing the enclave to perform dynamic memory allocation and, consequently, to be initialized faster (Silva et al., 2017). Other works seek to reduce the performance impact caused by context switching between the enclave and the application (Arnavtov

^a <https://orcid.org/0000-0003-2976-4533>

^b <https://orcid.org/0000-0003-2592-3664>

et al., 2016; Orenbach et al., 2017), and to extend the size of the SGX memory, so that it can cover all available main memory (Taassori et al., 2018). The common background in such approaches is the low-level code refinement and changes in the SGX software tools.

It is thus necessary to look for alternatives to reduce such performance overhead in the application and to ensure an efficient use of the SGX memory region, without compromising the security guarantees provided by SGX, and without changing its operating structures. Furthermore, the use of enclaves is recent and there is no clear position in the community on the correct way to incorporate them into applications, that is, on the most suitable programming models for building programs using enclaves.

In this paper, we propose two models of enclave management, enabling an efficient allocation of resources, aiming to reduce the use of SGX memory and the time spent by an application to require an enclave. Thus, the main contributions of this work are:

- Definition of enclave management models aiming to reduce the enclave creation overhead and the usage of enclave memory;
- A new approach to provide multiple enclaves to applications in a coordinated manner;
- A prototype that implements our approach validates it in real scenarios;
- Performance evaluation of our prototype, comparing it with previous works;
- Security assessment of the proposed enclave management models.

The remainder of this paper is organized as follows: Section 2 provides background information about the Intel Software Guard Extensions technology. Section 3 presents previous research closely related to our proposal, in the field of enclave management. Section 4 presents the enclave management models proposed in this work. Section 5 discusses the *Enclave as a Service* approach implemented in our prototype, which is described in Section 6. Performance evaluation of our solution is presented in Section 7 and the security assessment is discussed in Section 8. Finally, Section 9 concludes the paper and presents the future work.

2 INTEL SGX

The Intel *Software Guard Extensions (SGX)* architecture allows an application to be split into trusted and

untrusted components. The trusted part of the application runs in a secure environment called *enclave*, in which all instructions and data are kept in an encrypted region of the memory called *Processor Reserved Memory (PRM)* (McKeen et al., 2013). The main goal of the SGX architecture is to reduce the application's *Trusted Computing Base (TCB)* to a small piece of hardware and software.

To ensure data confidentiality and integrity, the SGX architecture provides new instructions, a new processor architecture, and a new execution model, which includes loading the enclave into the protected memory area, accessing resources via page table mapping and application scheduling inside the enclaves. After the application is loaded into an enclave, it is protected from any external access to the enclave, including access by applications that are in other enclaves. Attempts to make unauthorized changes to content within an enclave from outside are prevented by the hardware. While enclave data passes between the registers and other blocks of the processor, unauthorized access is prevented using the processor's own access control mechanisms. When data is written to memory, it is transparently encrypted and its integrity is maintained by avoiding memory probes or other techniques to view, modify, or replace data contained in the enclave (Costan and Devadas, 2016).

Memory encryption is performed using standard encryption algorithms, containing safeguards against replay attacks. Connecting the DRAM memory modules to another system will only give access to the data in its encrypted form. In addition, encryption keys are stored in registers inside the CPU, not accessible to external components, and they are changed randomly at every hibernation or system restart event.

2.1 Enclave Memory

Memory data protection is implemented by the *Enclave Page Cache (EPC)*, in which the memory pages and the SGX control structures are stored; this region is protected from external access by hardware. Data from different enclaves reside within the EPC; each enclave has its own control structure, called *SGX Enclave Control Structure (SECS)*, and each memory page in the EPC belongs to a single enclave. When an enclave requests access to the EPC, the processor decides whether to allow its access, managing security and access control within the EPC through a hardware structure called *Enclave Page Cache Map (EPCM)* (McKeen et al., 2013; Costan and Devadas, 2016).

The EPC storage in main memory is protected by encrypting the data to provide a defense against memory attacks. For that, a hardware unit called *Memory*

Encryption Engine (MEE) takes care of the encryption and data integrity while it is transferred between memory and the processor. A CPU with SGX support allows the BIOS to reserve a memory range called *Processor Reserved Memory (PRM)* to hold the EPC.

The CPU blocks any access to the PRM coming from external agents, treating these accesses as invalid address references. Likewise, attempts to access memory pages within an enclave by a process that is not running in that enclave are also treated as invalid references (Costan and Devadas, 2016).

2.2 Enclave Life Cycle

The enclave creation procedure consists of several steps: initialization of the enclave control structure, allocation of the memory pages in the EPC, loading the contents of the enclave for these pages, measurement of the contents of the enclave, and creation of an identifier for the enclave. Before starting the enclave creation, the process will already be residing in the main memory, being free for any inspection and analysis and, after it is loaded into the enclave, its data and code will be protected from any external access. The enclave life cycle is shown in Figure 1, which also shows the machine instructions responsible for managing the enclave.

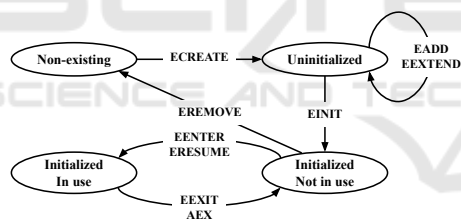


Figure 1: Enclave life cycle management instructions and state transition diagram (Costan and Devadas, 2016).

The `ECREATE` instruction starts the creation of the enclave and initializes the *SGX Enclave Control Structure (SECS)* which contains global information about the enclave. Memory pages are then added to the enclave using the `EADD` instruction. The `EEXTEND` instruction measures the contents of the enclave, which requires that all enclave code and data are already loaded into memory. Finally, the `EINIT` instruction completes the enclave creation process and creates its identity, allowing it to be used.

Execution can enter and leave an enclave using the SGX instructions `EENTER` and `EEXIT`, respectively. If the execution leaves an enclave due to some event or fault, the CPU will execute a routine called *Asynchronous Exit (AEX)*, which will save the enclave state, clear the registers and store the address of the instruction that generated the fault, allowing to resume

the enclave later by invoking the `ERESUME` instruction.

Finally, the enclave is destroyed using the `EREMOVE` instruction, which releases all EPC pages used by the enclave, ensuring that no logical processor is executing instructions within the EPC pages to be removed. The enclave is completely destroyed when the EPC page containing its SECS structure is released (McKeen et al., 2013; Costan and Devadas, 2016).

2.3 Attestation and Sealing

SGX provides *attestation* mechanisms, enabling another party to be confident that a software is securely running inside an enclave, in certified hardware, and was not tampered with. The attestation process can be performed locally or remotely. Local attestation enables that two enclaves in the same platform securely exchange data, attesting each other, by using symmetric encryption and authenticating the data. Remote attestation enables third parties to check the identity of an enclave running in a distinct SGX platform (Anati et al., 2013).

A *data sealing* procedure is also provided by SGX, allowing to securely store sensitive data in persistent storage. Each enclave can request to the CPU a unique key, called *sealing key*, derived from the enclave identity and the CPU itself. The sealing feature ensures that only the same enclave that sealed the data will be able to unseal them, and only when using the same CPU (Anati et al., 2013).

2.4 Performance and Memory Usage

Fisch *et al.* (Fisch et al., 2017) point out that the initialization of several enclaves by an application may impose a high computational cost, since, in principle, the enclave must have allocated all the memory it will need.

The size of the PRM should also be considered, since it is limited to 128 MB and, according to previous works (Shaon et al., 2017; Mofrad et al., 2017; Fuhry et al., 2017), only about 90 MB are effectively available for enclave data and code. This PRM limitation makes it necessary to think of alternatives for situations in which enclaves need to work with large amounts of data simultaneously. This may raise problems when there are multiple instances of the same enclave, or multiple enclaves in memory, forcing some of them to be brought to secondary memory in order to free space in the PRM. The memory swap/paging operation is supported by the Intel SGX architecture, but it obviously impacts performance.

3 RELATED WORK

Some strategies to improve the overall performance of applications that use SGX enclaves are presented in the literature. To reduce the performance impact caused by context switches, some solutions propose to provide a communication interface between the enclave and the application without the need for such switches, such as using threads running outside the enclave to receive the requests and then perform the system calls. Communication can be done through *Remote Procedure Call (RPC)* (Orenbach et al., 2017) or using a shared memory (Arnautov et al., 2016; Weisse et al., 2017). The performance improvement of system calls invocation by enclaves without context switching is investigated by (Tian et al., 2018), seeking to integrate this feature in the Intel SGX *Software Development Kit (SDK)*. Other works propose to improve I/O performance by reducing the number of enclave transitions (Svenningsson et al., 2021). An approach to reduce the encryption/decryption overhead when accessing data in memory is also proposed (Shimizu et al., 2019).

The cost of enclave creation is also covered in the literature, with approaches aimed at updating the binary code during runtime, by paging the enclave stack and allowing to load only parts of the enclave into memory (Krieter et al., 2019). Strategies to dynamically link libraries to enclaves and to allow dynamic loading of enclave code are also presented (Silva et al., 2017; Weichbrodt et al., 2021).

Approaches aimed at extending the PRM size are also presented. Taassori *et al.* (Taassori et al., 2018) propose a solution for the PRM to cover all available physical memory, by changing the EPCM structure and allowing to store non-sensitive memory pages in the EPC. Solutions that seek to store sensitive key-value data in primary memory also present approaches to use memory regions outside of PRM to store such data securely (Kim et al., 2019; Tang et al., 2019; Bailleu et al., 2019).

Programming paradigms are also used in order to reduce the impacts caused in the communication and synchronization of enclaves, which are needed in applications that use different enclaves to carry out their operations. A paradigm used in such context is the actor model, where each actor is self-contained and communicates with other actors through the exchange of messages, allowing the execution of parallel and non-blocking operations. This paradigm is used by (Sartakov et al., 2018), treating each enclave as an actor and reducing the need for context changes and synchronization between enclaves in parallel executions. Enclave replication in the cloud is addressed

by (Soriente et al., 2019), allowing seamless commissioning and decommissioning of SGX-based applications to achieve more efficiency in enclave management.

Despite there are solutions proposed in the literature that use programming models to overcome the performance issues of SGX, they are limited to specific enclaves and applications. We present in this work an architecture that deploys such models using an enclave management service, allowing multiple enclaves to be offered to multiple applications as logical resources, with their life cycle being managed by the proposed service.

4 ENCLAVE MANAGEMENT MODELS

The way the application manages the enclaves it uses may impact its performance, thus it is important to use efficient management techniques to minimize such impact. In addition, as previously mentioned, the limitations imposed by the SGX architecture in the use of PRM may increase this impact, due to the paging activity incurred in the execution of a large number of enclaves. Thus, management models that aim to reduce the number of enclave creations and simultaneously active enclaves in the system can bring a significant reduction in the performance overhead caused by the use of enclaves. This Section discusses two models of enclave management aiming at reducing such overhead.

4.1 Enclave Sharing

Some situations may present multiple instances of the same enclave, performing the same function and using mostly the same data. This is the case, for instance, of enclaves launched to perform user authentication, which use the same credential database; this may also happen in servers that receive a large number of requests to be processed in enclaves.

An alternative to reduce the number of instantiated enclaves is to share them by multiple applications or multiple instances of the same application, allowing the same instance of an enclave to treat requests from different sources and avoiding launching a new enclave for each request. The benefit of having shared long-life enclaves instead of exclusive short-life ones is twofold: it reduces the use of the PRM area, allowing more applications to have enclaves, and it avoids the costs of frequent enclave creation and sensitive data/code loading, which may be high, as discussed

in Section 2.4, improving the applications' response time.

Enclave sharing was adopted by (Will and Maziero, 2020) to centralize the handling of credentials data in an authentication system. The model proposed by the authors, using a client/server architecture, made it possible to apply SGX security guarantees without major performance impacts. A similar approach is used by (Karande et al., 2017) to centralize the manipulation of log files in operating systems.

4.2 Enclave Pool

The *pooling pattern*, defined by (Kircher and Jain, 2002), is a pattern that describes how the acquisition and release of expensive resources can be minimized by recycling the resources that are no longer needed. This pattern can be used in a context where systems continuously acquire and release the same or similar resources and need to meet high demands for scalability and efficiency, while seeking to ensure that system performance and stability are not compromised.

The object pool idea fits the limitation imposed by the SGX architecture in the PRM size, which ends up limiting the number of instances of enclaves that can reside in memory simultaneously, as well as with the cost of starting the enclaves, which increases in proportion to their size. Maintaining a pool of enclaves can decrease the response time of requests with frequent demands to enclaves, a situation that can occur in some implementations, as in the solution presented by (Brenner et al., 2017).

The use of an enclave pool is addressed by (Li et al., 2019), being applied in a multi-threaded web server context. In their proposal, each enclave on the server is linked to a thread responsible for direct communication with that enclave, avoiding problems resulting from race conditions within enclaves, in addition to a queue of requests to distribute each request to an enclave in the pool. Tasks to be performed in enclaves are queued and bound to available enclaves as soon as possible, maintaining control over the execution flow. In addition to the reduction in the impact resulting from an enclave creation for each new request, using an enclave pool allows a reduction in CPU usage, compared to the standard model of enclave use.

5 ENCLAVE AS A SERVICE APPROACH

An enclave can be seen as a component for the trusted execution of some sensitive routine, which is directly

linked to the application that uses it. On the other hand, the enclave can also be considered as a service provider for that application, for the execution of trusted routines, and for the provision of an encapsulated and protected environment for handling sensitive data.

The concept of offering resources or routines as a service is widespread in cloud computing environments, where it is denoted by the term *Everything as a Service* (XaaS). This concept allows integration between heterogeneous applications, with resources being packaged in services that are accessed by client applications. Services are fundamental items, totally independent from applications, allowing their use on-demand (Robison, 2008; Li and Wei, 2014).

In cloud environments, there are several resources delivered to applications in the form of services, such as software resources, with the concept of *Software as a Service* (SaaS), and hardware resources, with the concept of *Infrastructure as a Service* (IaaS). In the context of computer security, the *Security as a Service* (SECaaS) model is also presented, which provides, in the form of services, authentication and authorization mechanisms, intrusion detection, and verification of malicious software, among others. A wide classification of such services is presented by (Duan et al., 2015a) and (Duan et al., 2016). Despite being quite widespread and used in cloud environments, the XaaS paradigm is not restricted to these environments and can also be used locally, through the use of *daemon* processes to treat requests from system users or other processes (Duan et al., 2015b).

The XaaS paradigm key concept of unbinding resources from applications can be applied to the use of enclaves, offering them as a service to applications. Thus, an *Enclave as a Service* (EaaS) paradigm can be defined, which is characterized by a set of independent enclaves that are made available to different applications for the execution of trusted routines. This concept is already partially explored with the execution of secure microservices, which offer a trusted environment for the execution of small services in the cloud.

This paper proposes an approach to provide on-demand enclaves to applications, through an *enclave provider*, which creates and manages multiple enclaves in behalf of them. In this *Enclave as a Service* (EaaS) framework, applications only need to indicate the enclave they need, among a set of previously registered ones, and submit requests to it through the provider. All the enclave launching and management functions is performed by the provider.

6 ENCLAVE PROVIDER

The use of an enclave is restricted to the process that instantiated it; other processes are denied access when trying to make requests to a third-party enclave. Thus, to allow enclaves to be shared by several applications, it is necessary to include a mediator entity between the requesting applications and the shared enclaves. This mediator, called here as *enclave provider*, provides means for applications to communicate with the available enclaves safely and manages the life cycle of enclaves, relieving the applications of such task.

The enclave provider manages, in a coordinated manner, the request of N applications for M enclaves, according to the parameters contained in each request, whose details are described in Section 6.2. For that, the enclave provider centralizes all requests from applications in a broker, which is in charge of validating the request parameters, forwarding it to the corresponding enclave, and returning the corresponding response back to the requesting application. The proposed architecture, at a high level, is shown in Figure 2.

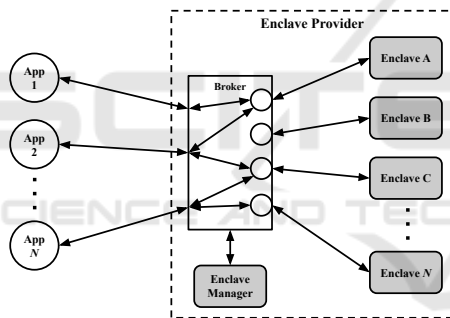


Figure 2: General architecture of the enclave provider, in which several applications communicate with different enclaves through a broker, which handles requests, forwards them to the corresponding enclave, and returns the result to the requesting application.

The sensitive information for the configuration and management of the enclave provider is kept sealed and manipulated within an *enclave manager*, ensuring its confidentiality and integrity. Such information includes the list of enclaves registered with the provider, with their respective configurations, and the list of active applications, enclaves, and the communication channels established between them. The architectural design details are described in the following sections.

6.1 Enclave Registration

The first step for an enclave to be made available for use by applications through the enclave provider is its registration on the platform. Enclave registration is necessary for the provider to become aware of the enclave and its additional settings so that it can be properly instantiated when requested.

The registered enclave will be identified by a unique name, which will be used by applications to make requests to the enclave. In this stage the criteria for the management of the enclave is defined, identifying whether the enclave will be managed as a pool or a shared instance among the applications, the number of client applications that the same enclave instance can serve, and whether one or more instances of that enclave should be created at the startup of the enclave provider. Also, the ECALLs available in that enclave are informed with their indexes, according to the order in which they appear in the *Enclave Definition Language* (EDL) file; finally, are defined the specific ECALLs to be executed when the enclave is instantiated or released.

Finally, the enclave provider performs the enclave validation procedure for the registration. This procedure first checks if there already is an enclave registered with the same requested name: if it exists, the registration is denied. After that, the enclave cryptographic hash is calculated and compared with the hash provided in the registration request: if they are not equal, the registration is denied. If both checks succeed, the enclave is registered in the provider and is made available for use by applications.

6.2 Communication

For an application to be able to use an enclave provided by the enclave provider, it should request an instance of that enclave to the enclave provider. This procedure is described in Figure 3 and consists of the following steps:

1. The application initiates a key agreement with the enclave provider, aiming to establish a secret key between both parties to encrypt the exchanged data. At the end of this step, the application and provider share a secret key SK_1 , which will be used to protect the subsequent communication between them;
2. A second key agreement is then performed between the application and the target enclave, to establish another secret key (SK_2) to protect the communication between them;
 - (a) The first step of this communication, between the application and the provider, is already en-

- encrypted using key SK_1 , with the provider decrypting the content of the request and collecting the necessary information to identify the target enclave;
- (b) The request is sent to the target enclave to proceed with the key agreement;
 - (c) The response from the enclave is returned to the provider, encrypted using SK_1 and sent back to the application;
3. The subsequent communications between the application and the enclave are protected with end-to-end encryption, with the necessary data for the provider being encrypted with the SK_1 key, and the data that will be sent to the enclave (payload) encrypted with the SK_2 key;
- (a) The provider uses the SK_1 key to decrypt the request header, in order to identify the target enclave;
 - (b) The request payload is sent to the enclave's ECALL function requested by the client application;
 - (c) The enclave's response is sent back to the application.

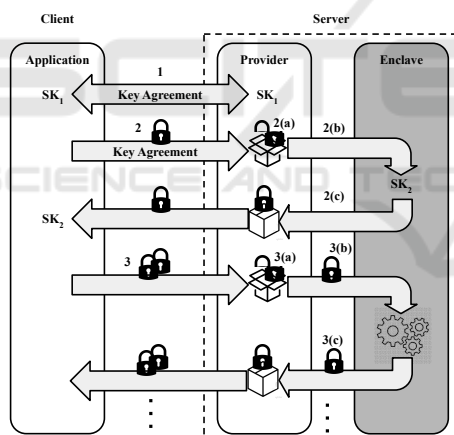


Figure 3: Communication between the application and the enclave, mediated by the enclave provider, with end-to-end encryption.

The request payload should not be seen by the provider, but it must be informed of which enclave is being requested and which method/action should be invoked on that enclave. To satisfy such requirements, the request data is protected by two encryption layers: the request payload, to be sent to the target enclave, is encrypted using the SK_2 key, maintaining the confidentiality of such data even for the provider; a header containing information for the provider is added to the request, and the whole data is then encrypted with the SK_1 key. This protects the request

confidentiality while allowing the provider to decrypt its header and manage the request.

A service request received by the provider should contain the requester identity, the target enclave, the ECALL to be invoked, and its parameters (payload). The application process identifier (PID) can be used as requester identity; the target enclave is defined by the *enclave identifier* (EID) received by the application at the end of the first key agreement; the ECALL to be invoked is defined by its name in the EDL file; finally, the payload is opaque data encrypted using SK_2 and passed as-is to the enclave. This data will allow the provider to make the function call to the target enclave and return the response to the requesting application. The response is protected using the same double-key schema and contains an opaque payload with the data returned by the enclave, when applicable, and a status indicating the success or failure in the execution of the requested operation by the provider.

6.3 Enclave Life Cycle

The enclave provider mediates the communication between the application and the enclave, but is also responsible for managing the life cycle of registered enclaves, seeking to maintain efficiency in the use of the resources required by each enclave. From the point of view of the requesting application, the use of enclaves mediated by the provider is similar to the use of an enclave directly by the application itself, maintaining the same steps of requesting the enclave, calling ECALLs, and releasing the enclave.

When the provider receives the request for an enclave, it verifies if there is already an active instance of the requested enclave. If there is no such active instance, the provider creates a new instance of that enclave. Otherwise, the provider verifies whether the maximum number of applications able to share the same enclave instance has been reached. If this number has not yet been reached, the provider returns the existing instance to the application, otherwise, a new instance of the enclave is created to attend the request.

The execution of ECALLs occurs as described in Section 6.2, with the request data being encapsulated, encrypted, and sent to the provider. It then decrypts its header to get the necessary information to identify the requested enclave and function, makes the requested call, and returns the response data back to the application.

The enclave provider also has a structure that maintains a list of all instantiated enclaves and which applications are using them, with each application receiving a unique identifier for each enclave it is using (EID). In this way, it is possible to validate the number

of client applications that are using a certain enclave instance and also to validate subsequent requests for applications to execute ECALLs.

6.4 Enclave Release

When an enclave is no longer needed, the application that requested it must indicate this to the provider. This operation is analogous to the call to the function `sgx_destroy_enclave` available at SGX API and, for this, the application makes a request to the enclave provider informing the identifier it holds.

When the provider receives the request to release an enclave by an application that was using it, the provider removes the entry corresponding to that application in its list of instantiated enclaves; that enclave instance will remain available for use by other applications. Even if there is no other application using the same instance of the enclave, that instance is still maintained for some time, being able to meet new requests.

In addition, the enclave provider must ensure responsible use of the enclaves: if an application requested an enclave instance and, after a long period of time, is no longer using it, the provider will remove its entry from the list of instances and, if there is no other application using that enclave instance, it is then removed.

7 PERFORMANCE EVALUATION

A performance analysis was carried out, to measure the impact caused by the use of the enclave provider, with either enclave sharing or pool, compared to enclaves instantiated and accessed directly an application. We run the performance tests in a Dell Inspiron 7460 laptop with the following settings: dual core 2.7 GHz Intel Core i7-7500U *Central Processing Unit (CPU)*, 16 GB RAM, 128 GB SSD and SGX enabled with 128 MB PRM size. Intel *TurboBoost*, *SpeedStep*, and *HyperThread* CPU extensions were disabled, to provide stable results. We used Ubuntu 18.04 LTS, kernel 4.15.0-112-generic, libdbus 1.12.20, Intel SGX SDK 2.9.101.2, and set the enclave heap size at 64 KB and the enclave stack size varying from 8 KB to 128 KB. The confidence interval width at 95% is 0.67% of the average. All times were measured using the *RDTSOP* instruction (Paoloni, 2010).

Figure 4 shows the average response time for requesting a given enclave, for 10,000 sequential requests with different enclave sizes (no ECALL is invoked). Note that the average response time for requests made directly by the application, without an

enclave provider, increases as enclave stack size increases, due to the need to statically allocate the necessary memory at the time of enclave initialization. Also, there is no significant difference in response times when using enclave sharing or pool approaches, they remain constant regardless of the enclave stack size.

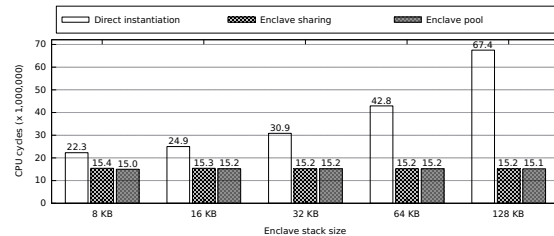


Figure 4: Average response time for enclave requests.

We also evaluated our solution with a real application, an authentication module of the *Pluggable Authentication Modules (PAM)* framework. We compared five distinct solutions:

- **Solution 1.** The PAM module itself, without any changes;
- **Solution 2.** The PAM module using SGX enclaves (UniSGX), as proposed by (Condé et al., 2018);
- **Solution 3.** The PAM module with an authentication server and enclave sharing, as proposed by (Will and Maziero, 2020);
- **Solution 4.** The PAM module with the enclave provider, which handles a shared enclave to validate the authentication requests;
- **Solution 5.** Same as Solution 4, adding the payload encryption and ensuring application-to-enclave confidentiality;

The results presented in Figure 5 show that the direct initialization of an enclave to validate the user's credentials (UniSGX solution, proposed by (Condé et al., 2018)) imposes a high performance cost. On the other hand, the use of an enclave sharing approach, such as the one proposed by (Will and Maziero, 2020), guarantees the security properties of SGX when manipulating the credentials file within an enclave, while having a virtually zero performance impact compared to using standard PAM.

The use of the enclave provider also does not have a significant performance impact, and maintains the same confidentiality assumptions of the data transported on the communication bus, since these are fully encrypted. Finally, since user credentials are strictly sensitive data, payload encryption ensures that

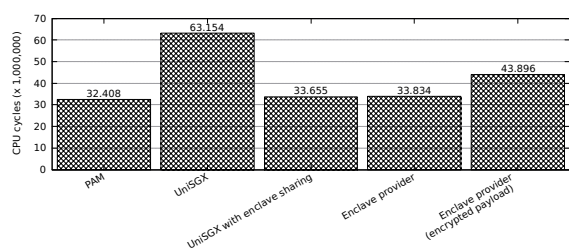


Figure 5: Response times for user authentication using PAM directly and the SGX-based solutions.

not even the enclave provider is aware of what is being passed between the client application and the requested enclave. Such operation adds a performance cost of approximately 30%, as a result of the data encryption and decryption operations and the additional key agreement operation.

In addition to the response times for sequential requests, we also seek to assess the behavior of the enclave provider to respond to multiple simultaneous requests. In this case, the response time for each request was evaluated, considering the enclave request and the execution of the ECALL, and the number of requests replied per second. For the tests, the enclave stack was set at 8 KB, to evaluate the shortest response time, and three different scenarios were considered:

- **Scenario 1.** 20 simultaneous instances of the client application are launched, each making 500 sequential requests to the enclave. The enclaves pool is configured with 20 instances;
- **Scenario 2.** 50 simultaneous instances of the client application are launched, each making 200 sequential requests to the enclave. The enclaves pool is configured with 20 instances;
- **Scenario 3.** 50 simultaneous instances of the client application are launched, each making 200 sequential requests to the enclave. The enclaves pool is configured with 50 instances;

Figure 6 shows the response times for each request. Note that in Scenario 1 the use of both enclave sharing and pool provide lower response time when compared to the direct initialization of the enclave by the application. Scenario 2 presents additional overhead to enclave sharing, due to the serialization imposed by the enclave provider to the incoming requests. This can be seen also in the enclave pool, mainly due to the pool under sizing. Finally, in Scenario 3, with the size of the enclave pool properly sized, the response time for each request is very similar to that presented when the application instantiates the enclave directly.

In addition to the response time for each request,

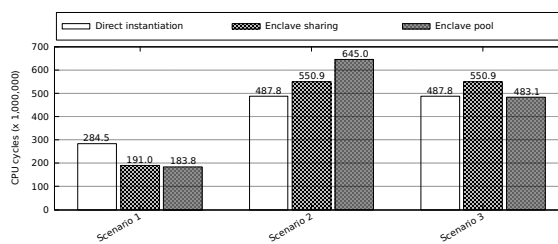


Figure 6: Average response time for each request (including enclave request and ECALL execution).

the average number of requests replied to per second in each scenario was also evaluated, with the results being presented in Figure 7. It is noted that, in Scenario 1, both enclave sharing and pool performed better than direct enclaves initialization. In Scenario 2, the enclave sharing approach maintained a good performance, whereas the enclave pool had a performance far below since the pool was undersized. With the correct sizing of the pool (Scenario 3), the enclave provider has a high response rate, again higher than direct initialization.

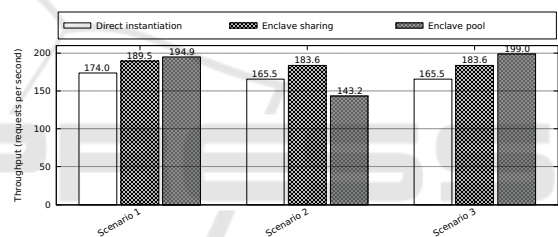


Figure 7: Enclave requests served per second.

The behavior of the enclave provider in the presence of multiple requests was also evaluated in a real application. Again, the PAM and the solutions proposed by (Condé et al., 2018) and (Will and Maziero, 2020) were used for this analysis. Two scenarios were considered:

- **Scenario 1.** 20 simultaneous application instances are launched, each instance makes 500 requests for user authentication;
- **Scenario 2.** 50 simultaneous application instances are launched, each instance makes 200 requests for user authentication;

The average response time for each request is shown in Figure 8. The solutions based on the enclave provider showed a response time higher than PAM and the enclave sharing solution proposed by (Will and Maziero, 2020), but still lower than the use of an enclave instantiated directly by the PAM authentication module, as proposed by (Condé et al., 2018). It is also noted that the payload encryption adds a small overhead on the performance of the enclave provider.

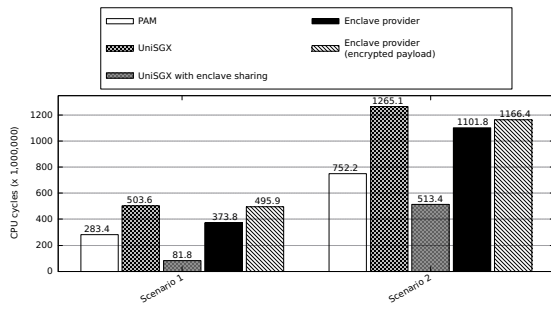


Figure 8: Average response time for each authentication process.

When analyzing the average number of requests handled per second, the enclave provider shows good performance, presenting rates higher than other solutions based on enclaves in Scenario 1, as presented in Figure 9. In Scenario 2, the enclave provider achieved a response rate equivalent to the enclave sharing solution described in (Will and Maziero, 2020) and, when added the payload encryption, demonstrated a response rate equivalent to direct initialization of the enclave, as proposed by (Condé et al., 2018). Such results demonstrate the robustness of the enclave provider in the demand for several simultaneous requests.

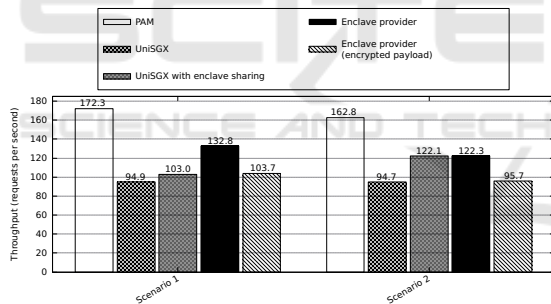


Figure 9: Rate of requests replied per second for user authentication using PAM.

8 SECURITY ASSESSMENT

Our threat model takes into account that the attacker can monitor the communication mechanism between applications and can collect the data exchanged between them. Thus, sensitive data exchanged between applications and the enclave provider must be guaranteed confidentiality. Full control of the communication mechanism by the attacker, allowing it to change or replace the data that are transmitted, is considered out of scope.

The enclaves managed by the provider are previously registered, and the attacker can access the file

that contains this information and tamper with or replace its contents. In this case, protection mechanisms must be applied to guarantee the confidentiality and integrity of this content. Additionally, the attacker can substitute the code files for the registered enclaves, making it necessary to use mechanisms to validate these files before loading them.

Internal data structures of the enclave provider can be manipulated by an attacker, aiming to seize requests from legitimate applications and forward them to malicious enclaves. Such structures must be kept in a protected area, preventing them from being deliberately tampered with in memory. Finally, client application vulnerabilities are considered out of scope in our threat model.

All sensitive data structures for the execution of the enclave provider are maintained and handled by an enclave manager, thus using the confidentiality and integrity mechanisms provided by SGX. The enclave manager is also responsible for receiving and distributing requests from client applications, keeping records of which enclaves are instantiated, which applications are making requests, and which instances of enclaves are associated with these applications.

The enclaves handled by the provider are previously registered, and such records are maintained and updated by the enclave manager, being stored in secondary memory in encrypted form, using the SGX sealing feature. Thus, the records can only be read by the manager enclave itself, and only on the platform used to seal the data. In addition, any change in the encrypted data is identified by the authentication mechanisms contained in the sealing procedure itself.

Each enclave registered in the provider is accompanied by an SHA-256 cryptographic hash, which aims to validate the content of the enclave before it is loaded. Thus, the enclave provider validates the cryptographic hash obtained from the content of the enclave (including code and data) with that informed in the registration data, ensuring that the code and data of the enclave have not been altered or replaced. This verification is carried out both when the enclave is registered and when creating each of its instances, which allows one to identify the changes made to the enclave after its registration. The entire verification procedure is carried out within the limits of the managing enclave, avoiding the manipulation of such data by unauthorized entities. After creating the instances, the mechanisms provided by SGX guarantee the integrity of the data and code of the enclave.

The communication between the client application and the enclave provider is fully encrypted, and the enclave manager is responsible for handling session keys and decrypting incoming requests. The ses-

sion keys are agreed between the client application and the managing enclave through an elliptic curve Diffie-Hellman (ECDH) key agreement protocol, using the curve X25519 proposed by (Bernstein, 2006). The use of the Diffie-Hellman protocol allows the definition of an encryption key between the two parties through an insecure channel, preventing an attacker from obtaining this key.

The requests sent to the enclave provider are encrypted using a symmetric AES-CTR encryption algorithm, using the 128-bit key agreed between the parties. This key is used to encrypt all communication between the application and the enclave provider, including the responses sent by the provider. The application can reset the encryption key at any time by initiating a new key agreement with the enclave provider. The encryption of messages exchanged between the parties ensures that if an attacker is connected to the message bus and collects the information exchanged, she will not have direct access to it, requiring the application of techniques to obtain the key used for encrypting the data. Even using high computational power, the process of obtaining the key is extremely expensive, making it unfeasible.

The procedures for decrypting the request and encrypting the response to be sent to the application occur within the manager enclave, which manipulates the session keys. In this way, the encryption keys are always protected by the enclave provider, ensuring that they are not accessed by unauthorized entities.

Client applications can add an extra layer of security by making a key agreement directly with the requested enclave, thus enabling end-to-end encryption of the data sent to the enclave. This possibility makes the data exchanged between the application and the enclave totally opaque to the enclave provider, in addition to adding an extra obstacle to an attacker who may be collecting data in transit.

The client-server architecture also implies a single point of failure: the server application. Thus, the enclave provider must be resilient and capable of recovering from failures, to avoid denial of service attacks, which prevent legitimate client applications from accessing the requested enclaves.

Finally, in addition to the security issues that should be considered for the enclave provider in general, we must consider the implications of each proposed enclave management model. It should be noted, however, that the management mode is a choice made by the developer of the enclave, and he must take into account the balance between performance and the expected security guarantees.

9 CONCLUSION

This paper proposes some programming models for building applications using Intel SGX enclaves, to reduce the performance impact of this technology and optimize resource utilization. Two different models of enclave management were described: *sharing* and *pool*. Each model has its own characteristics and can be used in different contexts. Considering these models, an architecture was proposed for an enclave provider, which offers each enclave as a software resource to the applications and makes it available to client applications in the form of a service, decoupling the enclave from the application that uses it and allowing centralized management and efficient use of enclave instances. After the specification of the enclave provider, a prototype was developed to validate the proposal.

As future work, we intend to mitigate the problem of under sizing the enclave pool by using more efficient policies in releasing enclaves, causing the enclave provider to monitor pool usage and create new instances in the background when necessary. Other enclave management models may be added to the provider, ensuring a greater diversity of models and meeting other types of demand.

Extensions to the presented solution, in order to work with requests from IoT devices and to run the provider in a cloud environment, are also being studied, including operations that depend on the state of the enclave, by using enclave migration techniques.

REFERENCES

- Anati, I., Gueron, S., Johnson, S. P., and Scarlata, V. R. (2013). Innovative technology for CPU based attestation and sealing. In *Intl Wksp on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel. ACM.
- Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O’Keeffe, D., Stillwell, M., et al. (2016). SCONE: Secure Linux containers with Intel SGX. In *Sym on Operating Systems Design and Implementation*, Savannah, GA, USA. USENIX Association.
- Bailleu, M., Thalheim, J., Bhatotia, P., Fetzer, C., Honda, M., and Vaswani, K. (2019). SPEICHER: Securing LSM-based key-value stores using shielded execution. In *Conf on File and Storage Technologies*, Boston, MA, USA. USENIX Association.
- Bernstein, D. J. (2006). Curve25519: New Diffie-Hellman speed records. In *Intl Wksp on Public Key Cryptography*, New York, NY, USA. Springer.
- Brenner, S., Hundt, T., Mazzeo, G., and Kapitza, R. (2017). Secure cloud micro services using Intel SGX. In *Intl*

- Conf on Distributed Applications and Interoperable Systems*, Neuchatel, Switzerland. Springer.
- Condé, R. C. R., Maziero, C. A., and Will, N. C. (2018). Using Intel SGX to protect authentication credentials in an untrusted operating system. In *Sym on Computers and Communications*, Natal, RN, Brazil. IEEE.
- Costan, V. and Devadas, S. (2016). Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86.
- Duan, Y., Cao, Y., and Sun, X. (2015a). Various “aaS” of Everything as a Service. In *Intl Conf on Software Engineering*, Takamatsu, Japan. IEEE.
- Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C., and Hu, B. (2015b). Everything as a Service (XaaS) on the cloud: Origins, current and future trends. In *Intl Conf on Cloud Computing*, New York, NY, USA. IEEE.
- Duan, Y., Sun, X., Longo, A., Lin, Z., and Wan, S. (2016). Sorting terms of “aaS” of everything as a service. *Int. J. Networked Distrib. Comput.*
- Fisch, B. A., Vinayagamurthy, D., Boneh, D., and Gorbunov, S. (2017). Iron: Functional encryption using Intel SGX. In *Conf on Computer and Communications Security*, Dallas, TX, USA. ACM.
- Fuhr, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F., and Sadeghi, A.-R. (2017). HardIDX: Practical and secure index with SGX. In *Conf on Data and Applications Security and Privacy*, Philadelphia, PA, USA. Springer.
- Karande, V., Bauman, E., Lin, Z., and Khan, L. (2017). SGX-Log: Securing system logs with SGX. In *Asia Conf on Computer and Communications Security*, Abu Dhabi, UAE. ACM.
- Kim, T., Park, J., Woo, J., Jeon, S., and Huh, J. (2019). ShieldStore: Shielded in-memory key-value storage with SGX. In *EuroSys Conf*, Dresden, Germany. ACM.
- Kircher, M. and Jain, P. (2002). Pooling pattern. In *Euro Conf on Pattern Languages of Programs*, Irsee, Germany. UVK.
- Krieter, S., Thiem, T., and Leich, T. (2019). Using dynamic software product lines to implement adaptive SGX-enabled systems. In *Intl Wksp on Variability Modelling of Software-Intensive Systems*, Leuven, Belgium. ACM.
- Li, D., Lin, R., Tang, L., Liu, H., and Tang, Y. (2019). SGX-Pool: Improving the performance of enclave creation in the cloud. *Trans. Emerging Telecommun. Technol.*
- Li, G. and Wei, M. (2014). Everything-as-a-service platform for on-demand virtual enterprises. *Inf. Syst. Front.*
- McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. (2013). Innovative instructions and software model for isolated execution. In *Intl Wksp on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel. ACM.
- Mofrad, M. H., Lee, A., and Gray, S. L. (2017). Leveraging Intel SGX to create a nondisclosure cryptographic library. Preprint arXiv:1705.04706.
- Orenbach, M., Lifshits, P., Minkin, M., and Silberstein, M. (2017). Eleos: Exitless OS services for SGX enclaves. In *Euro Conf on Computer Systems*, Belgrade, Serbia. ACM.
- Paoloni, G. (2010). How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation*.
- Robison, S. (2008). The next wave: Everything as a Service. *Hewlett-Packard Company*.
- Sartakov, V. A., Brenner, S., Ben Mokhtar, S., Bouchenak, S., Thomas, G., and Kapitza, R. (2018). EActors: Fast and flexible trusted computing using SGX. In *Intl Middleware Conf*, Rennes, France. ACM.
- Shaon, F., Kantarcioglu, M., Lin, Z., and Khan, L. (2017). SGX-BigMatrix: A practical encrypted data analytic framework with trusted processors. In *Conf on Computer and Communications Security*, Dallas, TX, USA. ACM.
- Shimizu, A., Townley, D., Joshi, M., and Ponomarev, D. (2019). EA-PLRU: Enclave-aware cache replacement. In *Intl Wksp on Hardware and Architectural Support for Security and Privacy*, Phoenix, AZ, USA. ACM.
- Silva, R., Barbosa, P., and Brito, A. (2017). DynSGX: A privacy preserving toolset for dynamically loading functions into Intel SGX enclaves. In *Intl Conf on Cloud Computing Technology and Science*, Hong Kong, China. IEEE.
- Soriente, C., Karame, G., Li, W., and Fedorov, S. (2019). ReplicaTEE: Enabling seamless replication of SGX enclaves in the cloud. In *Euro Sym on Security and Privacy*, Stockholm, Sweden. IEEE.
- Svenningsson, J., Paladi, N., and Vahidi, A. (2021). Faster enclave transitions for IO-intensive network applications. In *Wksp on Secure Programmable Network Infrastructure*, Virtual Event. ACM.
- Taassori, M., Shafiee, A., and Balasubramonian, R. (2018). VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *Intl Conf on Architectural Support for Programming Languages and Operating Systems*, Williamsburg, VA, USA. ACM.
- Tang, Y., Li, K., and Chen, J. (2019). Authenticated LSM trees with minimal trust. In *Intl Conf on Security and Privacy in Communication Systems*, Orlando, VA, USA. Springer.
- Tian, H., Zhang, Q., Yan, S., Rudnitsky, A., Shacham, L., Yariv, R., and Milshten, N. (2018). Switchless calls made practical in Intel SGX. In *Wksp on System Software for Trusted Execution*, Toronto, ON, Canada. ACM.
- Weichbrodt, N., Heinemann, J., Almstedt, L., Aublin, P.-L., and Kapitza, R. (2021). sgx-dl: Dynamic loading and hot-patching for secure applications: Experience paper. In *Intl Middleware Conf*, Québec City, QC, Canada. ACM.
- Weisse, O., Bertacco, V., and Austin, T. (2017). Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *Intl Sym on Computer Architecture*, Toronto, ON, Canada. ACM.
- Will, N. C. and Maziero, C. A. (2020). Using a shared SGX enclave in the UNIX PAM authentication service. In *Intl Systems Conf*, Montreal, QC, Canada. IEEE.