# Using a Shared SGX Enclave in the UNIX PAM Authentication Service

Newton C. Will, Carlos A. Maziero
Computer Science Department
Federal University of Paraná State
Curitiba, Brazil, 81530–015
E-mail: {ncwill, maziero}@inf.ufpr.br

***Abstract*—Confidentiality in the storage and handling of sensitive data is a central concern in computing security; one of the most sensitive data in computer systems is users' credentials. To ensure the confidentiality and integrity of sensitive data, developers can use a Trusted Execution Environment (TEE). One of such TEE is Intel Software Guard Extensions (SGX), which reduces the trusted computing base to a hardware/software concept called *enclave*. However, using SGX enclaves usually incurs in a performance impact in the application execution. In this paper we propose an enclave sharing approach to reduce the performance overhead in scenarios where multiple enclaves handle the same data. To evaluate this approach, we implemented a SGX-secured OS authentication service. Three prototypes were built, considering distinct concerns about security and performance. Results show that this approach can be used in high demand environments, presenting a small overhead.**

***Index Terms*—Trusted execution, programming models, software architecture, design patterns, performance.**

## I. Introduction

Data confidentiality is very important to computer users, who store their data in devices or cloud services. Access to such devices or services is commonly provided through an authentication procedure to validate the user's identity. Obviously, user credentials must be securely stored and handled, but credential leaks are a major problem today [1].

To improve the confidentiality and integrity of sensitive data, Intel released the *Software Guard Extensions* (SGX) architecture, which provides an isolated trusted environment to protect data and code in an encrypted memory area, which is handled solely by the CPU [2]. The Intel SGX architecture reduces the Trusted Computing Base (TCB) to a piece of software and hardware, called *enclave*, but imposes an overhead like any other Trusted Execution Environment (TEE).

In this paper we propose an enclave sharing approach to reduce the performance overhead in scenarios where multiple enclaves manipulate the same data. This approach is based on using a client/server architecture to share the enclaves. To evaluate our proposal, we implemented a SGX-secured OS authentication service as a proof of concept (PoC). Three prototypes of this service were built, with distinct concerns about the security and the performance of the solution. Results show that this approach can be used in high demand environments, posing a reduced overhead.

The remainder of this paper is organized as follows: Section II provides background information about Intel Software Guard Extensions and the Linux authentication framework. Section III presents previous research closely related to our proposal. Section IV discusses performance impacts caused by the SGX architecture. Section V presents the enclave sharing approach. In order to validate the approach, three prototypes are presented in Section VI as a proof of concept. Section VII presents the performance evaluation and security assessment. Section VIII concludes the paper.

## II. Background

This Section presents a brief description of important concepts used in this work, such as the Intel Software Guard Extensions and the Linux authentication framework.

### A. Intel SGX

Intel Software Guard Extensions (SGX) is an extension to the instruction set of the x86 architecture that allows applications to run in a protected memory area, called *Enclave*, which contains code and application data [2]. An enclave is an encrypted area in the address space of the application that ensures confidentiality and integrity of the data in it. Such data cannot be accessed by malware nor other software with high execution privileges, such as virtual machine monitors, BIOS, and the operating system [3]. This encrypted area is called *Processor Reserved Memory* (PRM), and its maximum size is defined in BIOS setup, being limited to 128 MB.

Each enclave has a self-signed certificate provided by the enclave code author, allowing the Intel SGX architecture to detect if any part of the enclave content was tampered with; this allows an enclave to prove that it was properly loaded and can be trusted. The *Enclave Measurement* is a unique 256 bits hash code that identifies the code and the initial data to be placed in an enclave, as well as the order and the position in which they should be placed and the security properties of their pages. A change in any of these variables will result in a different measurement [4].

The main goal of SGX is to reduce the Trusted Computing Base (TCB) of a computing system. The TCB reduction incurs in having fewer points of failure in the trusted part of the application, resulting in a safer software. For that, the developer should define which components of the software will be placed

978-1-7281-5365-6/20/$31.00 ©2020 IEEE

inside the enclave and which ones can be put outside. An enclaved software component should then provide a set of methods/functions available to the untrusted software components as *Enclave Calls* (ECALLs); also, services provided by the untrusted components to the enclave should be defined as *Outside calls* (OCALLs). All such functions should be defined by the developer [4].

To enable two or more enclaves to share data securely, the SGX architecture also supports *attestation*, which allows an enclave to prove its identity, authenticity, and integrity to another one. Attestation can be done either locally, between enclaves running on the same platform, or remotely, for enclaves running on distinct platforms [5].

Another mechanism provided by Intel SGX is *sealing*. When an enclave is running, its data confidentiality is guaranteed by keeping it encrypted in the PRM, but the data will be lost when the enclave is destroyed. The sealing procedure allows to store that data in a persistent storage, encrypted using a *sealing key*. This key can be derived from the enclave measure or the author signature.

### B. Linux Authentication

In its simplest form, Linux systems use two files to store users' information: `/etc/passwd` and `/etc/shadow`. The `/etc/passwd` file stores user information required for normal system operation: user IDs, group IDs, home directories, user shells, etc. The `/etc/shadow` file stores encrypted passwords, as well as management information, like expiration dates, etc.; it is readable only by the *root* user.

To check a user credential, applications do not access those files directly, but use the *Pluggable Authentication Modules* (PAM) framework. PAM uses dynamically loaded modules to implement mechanisms for authenticating users using information stored in files or authentication servers. In addition, PAM can also address other issues such as account, session, and password management [6]. The PAM framework also supports *stacking*, in order to support multiple authentication mechanisms. Thus, when the PAM API is called, the stack of modules is invoked in the selected order and the result of the operation is returned to the caller.

PAM reduces complexity in the authentication procedure, allowing the system administrator to use the same database in distinct authentication contexts; it also allows to use distinct underlying authentication mechanisms, transparently [7].

## III. Protecting OS components with SGX

The SGX architecture was launched in 2015, as part of the $6^{th}$ generation of Intel Core processors (*Skylake*). Since its announcement, several works were developed using it. One of the areas in which the SGX architecture is being used is to increase security in subsystems or modules of existing operating systems.

The paper [8] describes *SCONE*, which uses SGX in operating system containers, allowing *Docker* to support the secure environment provided by SGX. This approach is important because containers provide higher I/O performance and faster start up compared to virtual machines, but potentially offer a weaker isolation between containers and the host.

In [9] the authors presents a security architecture that leverages Intel SGX to protect security-sensitive computations inside unikernels. The authors obtained a minimum overhead in the enclave creation and OCALLs. The paper [10] propose a system for running Linux binaries on Intel SGX enclaves, containing a complete library operating system within the enclave, including file system and network stacks. Other similar approaches aim the creation of secure library OSes [11] and micro-containers using SGX [12].

Another solution is proposed by [13], which uses SGX to isolate Linux kernel modules into enclaves, preventing a failure in a kernel module from spreading to the rest of the system. To do this, portions of the module are migrated to user space, since enclaves cannot be run in kernel mode. The enclave is implemented by a daemon in the user space that instantiates and accesses the enclave.

In [14] the authors use SGX to maintain the integrity of the log files of an operating system. They use a client/server architecture, where the client is a log request component, which issues log messages, while the log server executes secure log services. The server is divided into two components: the trusted part, that handles sensitive data, and the untrusted part. In this way, several clients can issue requests to the log server, which in turn uses an enclave to store the logs.

In [15] the authors uses the sealing feature provided by the SGX architecture to encrypt the contents of the credential file (`/etc/shadow`) used by the PAM authentication framework (*Pluggable Authentication Module*) on Linux systems, presenting the *UniSGX* solution. The authors modified the PAM module `pam_unix.so`, allowing it to use a sealed credential file, using an enclave to verify the provided credentials.

Fig. 1 shows the control flow of the authentication procedure in UniSGX. When initiating the authentication procedure, the requesting application interacts with PAM to inform the credentials or to request the user to provide them. Upon reception of the credentials, PAM initiates the UniSGX module, which instantiates an enclave to receive the credentials and to validate them, using the sealed shadow file.

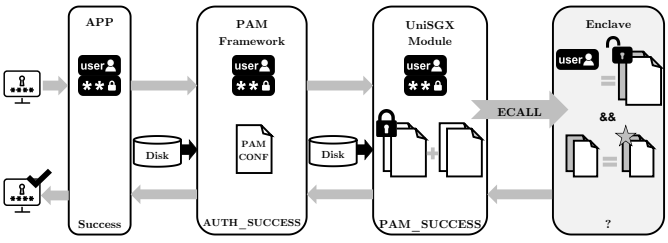


Fig. 1. Authentication using the UniSGX module [15].

## IV. SGX Performance and Limitations

Like other security solutions, the protection provided by SGX imposes a performance penalty. In [16], the authors observed that the initialization of several enclaves by the

application may have a high computational cost, since at first the enclave should allocate all the memory it will need. [17] presents some concerns about the creation and use of enclaves, in terms of performance, and makes recommendations about their use, such as provisioning enclaves, creating a enclave pool, and defining the maximum size of an enclave.

In [18] the authors analyse the overhead caused by PRM content ciphering procedure, the transition between the enclave and the application through ECALLs and OCALLs, and the encryption operations within an enclave.

PRM size should also be considered, as it is currently limited to 128 MB, of which only 90 MB being effectively available for enclaved data and code [19]. This limitation makes necessary to devise alternatives for situations where enclaves need to handle large amounts of data simultaneously, and may have some consequences when there are several instances of the same enclave or several enclaves in memory. Memory swap operation is supported by the Intel SGX architecture, but obviously brings an overhead.

## V. The Enclave Sharing Approach

The security provided by the SGX architecture has two limitations that should be taken into account when designing and developing SGX-secured applications: the PRM size limit and the cost of initiating an enclave. The impact of such limitations can be minimized by using efficient programming models and design patterns, in order to reduce the PRM usage and the number of enclave needed in the application.

Enclave sharing is discussed by [20]; they argue that the Intel SGX isolation primitives make it difficult to share code or data between enclaves, as there is no trusted shared memory. The authors highlight that, especially in library OSes, there are good reasons for sharing code, such as to reduce memory usage, to implement shared buffers, or to pass zero-copy messages. Beyond library OSes, other applications may also take advantage of an enclave sharing model.

### A. Design Patterns

Patterns are proven solutions to recurring design problems and generally describe pervasive structures that join modules produced by common design methods. Using design patterns, a system can be specified by describing the components that constitute it, their responsibilities and relationships with each other, and how they collaborate with each other.

Patterns are present in various levels in software development. Some patterns aid in structuring the software or its subsystems, others support the refinement of these subsystems and their components, as well as the relationships between them. Finally, there are also patterns that help implementing specific aspects of design in a specific programming language [21].

There are some design decisions associated to the use of enclaves, such as the definition of the programming model used to instantiate and manage the life cycle of the enclave, the choice of which data will be handled within the enclave, and when the application will communicate with the enclave.

The use of design patterns with SGX enclaves is already covered in other works, especially in cloud environments, such as [22] that use an enclave pool to assign/revoke each enclave on demand, thus avoiding its cost of dynamic creation and destruction. Another proposal is an autonomous membership service for enclave applications, providing elasticity for cloud applications, with automatic enclave commission and decommission [23].

In other situations, we can have multiple instances of the same enclave, running the same functions and using the same data, such as authentication requests or log writing. This implies in a high occupation of the PRM area, with a high number of enclaves performing the same task at the same time or using the same data. Moreover, if the enclave performs a short task, the time spent to create it and to load the data inside it may incur in a performance overhead. These problems can be mitigated by using appropriate design patterns in the application architecture and enclave life cycle management.

### B. The Proposed Architecture

The problem of having multiple instances of the same enclave manipulating the same data can be recurrent in computing systems that handles such data in a centralized way, especially on a server process that receives concurrent requests.

In this scenario, an enclave sharing approach may be useful, having a single enclave that manages the sensitive data and serves several applications simultaneously. The enclave sharing is designed to reduce the amount of memory required from the PRM to perform these tasks. Another aspect is that, by keeping an instance of the enclave active for later use by other applications, the enclave initialization time for the subsequent requests is saved, which brings an average performance gain in a large sequence of requests.

It should be noticed that the enclave sharing implies in extra cares about data leakage and confidentiality, since the several instances will be accessing the same enclave. The enclave code developer must ensure that no sensitive data from an application is shared with another one. Enclave sharing cannot be applied in situations where each client request must be processed using a different data set, as this could cause information leakage problems among client processes.

An approach to implement this enclave management model considers a client/server architecture, running in the same platform or in a distributed system. The server process encapsulates the enclave and is responsible for instantiating and managing its life cycle, receiving the requests coming from client applications, forwarding them to the enclave, receiving the response from the enclave, and sending it back to the requesting processes. This process must also queue incoming requests to allow controlled access to the enclave and to prevent information leaking among the different requests. A high-level view of the architecture is shown in Fig. 2.

The server process must also ensure the enclave availability to the requesting applications and may discard the enclave when it is no more needed. This process, called hereafter

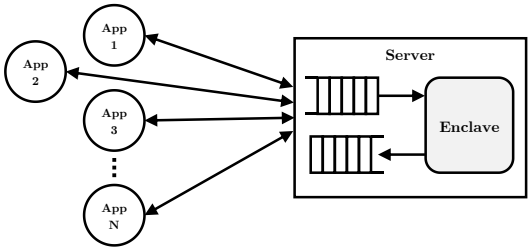


Fig. 2. Enclave sharing architecture overview.

the *enclave provider*, is necessary, as the SGX architecture limits the enclave calls only to the process that instantiates the enclave.

Communication between the applications and the enclave provider must be carried out using an IPC (*Inter-Process Communication*) mechanism and techniques to ensure the integrity and confidentiality of the transmitted data. This can be done by establishing a secure communication channel between the client and the enclave provider and using robust encryption algorithms, with predefined keys or a key agreement protocol. Also, both client applications and enclave provider can perform an enclave attestation procedure before transferring data, to ensure that the client process is running in the same platform or that the client is trustworthy (this process will be described in Section VI-C).

It is also necessary to use mechanisms for the treatment of race conditions in concurrent requests to the enclave, in order to maintain the integrity of the data in memory, especially in data updates. Outside the enclave, this can be achieved using semaphores to queue requisitions coming from clients to the server. Inside the enclave, this can be performed in a similar way by using spin locks, or by allowing only one thread to enter the enclave each time.

### C. Impacts

The use of a shared enclave between several applications brings considerable benefits to the execution of such applications, mainly given a better performance in the execution of sensitive tasks using the enclave. The use of a single enclave containing the sensitive data to be manipulated drastically reduces the processing required in the enclave creation and in the data loading, consequently reducing the time spent in the execution of the task, when taking into account a time interval with several requests. This is due to the fact that, after the first request, the enclave will already be instantiated and possibly the necessary data will already be loaded in the enclave. Both enclave creation and data loading into the enclave have a considerable impact on performance [17], [18].

In addition, in a scenario where multiple application instances are expected to be running and manipulating the same sensitive data over a given time interval, applying the enclave sharing model will also reduce the PRM usage. Such a reduction occurs due to the use of a single instance of the enclave containing the data to be manipulated, instead of several instances of the same enclave having the same data loaded in memory. This is important, as the PRM size is limited to 128 MB for all enclaves.

## VI. Proof of Concept

To assess the proposed enclave sharing approach, we implemented an authentication service based on the Pluggable Authentication Modules and the UniSGX solution [15], using a client/server architecture to reduce the performance impact caused by the creation of enclave instances. We used D-Bus (Desktop Message Bus) [24] as an IPC mechanism, as it is convenient and easily found in Linux platforms. Three implementations are presented, with distinct performance expectations and security concerns.

The common point in the three implementations is the existence of a server process that has the purpose of managing the life cycle of the enclave that handles the sealed shadow file. At the first request to the server process, the enclave is created and the shadow file is loaded into the enclave and then unsealed. Using the contents of the unsealed shadow file, the enclave satisfies the successive authentication requests coming from the PAM module.

### A. Implementation A - Hard-coded Keys

In this implementation, a separate process (the enclave provider) instantiates an enclave to perform the shadow file access and to satisfy authentication requests. The UniSGX PAM module passes the user/password data to the enclave provider through a D-Bus communication session. The data sent is encrypted by an 128 bit AES-CTR algorithm, in order to provide data confidentiality. The encryption key is hard-coded in the client and the enclave, and creates a secure communication channel between the client and the enclave. The provider will only receive and forward the encrypted data to the enclave.

For the first authentication request, the enclave provider creates an enclave instance and loads the shadow file into it. Then, it makes an ECALL to send the authentication request data to the enclave. The enclave will check the credentials, compare to those contained in the shadow file, and return the request result. This result will be then sent back to the UniSGX module through D-Bus, as shown in Fig. 3. Successive requests will use the enclave already instantiated.

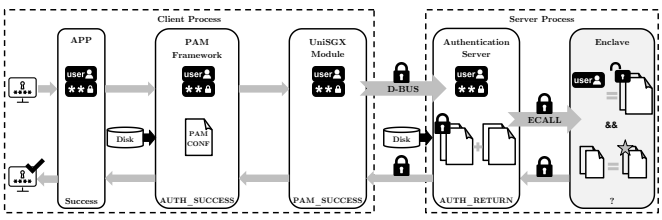


Fig. 3. Implementation A - Authentication control flow using the UniSGX Authentication Service.

This implementation is only intended to validate the communication between clients and the provider, and to measure

the overhead caused by the key agreement process present in implementation B.

### B. Implementation B - Key Agreement

To improve implementation A, we replaced the hard-coded keys by a more secure key agreement protocol, to define a session key for communication. In this implementation, the first step is to perform an Elliptic-Curve Diffie–Hellman (ECDH) key agreement scheme based on Curve25519 [25]. After the session key is defined between the client and the provider, interactions occur in the same way as presented in implementation A (see Fig. 4).

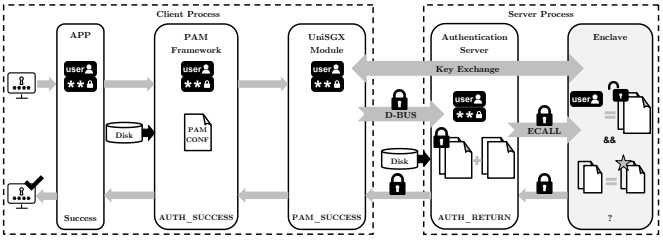


Fig. 4. Implementation B - Authentication control flow using the UniSGX Authentication Service with key agreement.

### C. Implementation C - Attestation

In some cases, it is important not only to ensure secure communication between clients and the provider, but also to ensure that the client process was properly established on the same platform as the provider, or even that the client process is running in a SGX enabled device.

In order to verify whether the UniSGX client process is running in the same platform as the server process, local attestation can be used. To achieve this, an enclave should be created at the client process, to perform the attestation and establish the communication with the server enclave. Fig. 5 presents this approach.

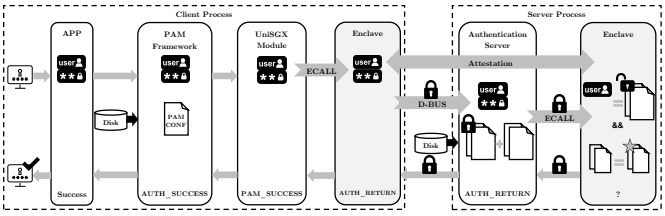


Fig. 5. Implementation C - Authentication control flow using the UniSGX Authentication Service with enclave attestation.

The client enclave initiates the attestation procedure using an unencrypted channel, aiming at the mutual validation of the client and the provider enclaves. In the attestation procedure, both enclaves establish a 128 bits encryption key using a Diffie–Hellman key agreement, which will be used to encrypt the content of exchanged messages. Thus, the user name and password are encrypted by the client enclave and decrypted only inside the server enclave, no clear data leaving the boundaries of the enclaves. The AES-GCM algorithm used in the encryption procedure also ensures data authenticity.

## VII. Evaluation

In order to evaluate the proposed approach, this section brings an analysis of the performance impact and the security assessment of the implementations presented in Section VI.

### A. Performance Evaluation

As presented in [15], enclave initialization and sealed data loading into it impose a high overhead to the UniSGX solution. By using an enclave sharing approach, this overhead can be reduced.

To evaluate the performance overhead of our solution, we measure the time spent in the authentication procedure and compare the three implementations with the original PAM Auth and the UniSGX modules [15]. We considered three shadow file sizes (5 KB, 100 KB, and 500 KB), sealed by SGX and with passwords hashed by the MD5 algorithm. 1,000 authentication requests were performed in each scenario. Times were measured as described in [15]. Experiments were run 10 times each; coefficients of variation observed in the results are under 5%.

Tests were run on a Dell Inspiron 7460 laptop, dual-core 2.7 GHz Intel Core i7-7500U CPU, 16 GB RAM, 1 TB hard drive, 128 GB SSD, SGX enabled with 128 MB PRM size, running Ubuntu 18.04 LTS, kernel 4.15.0-55-generic. Intel TurboBoost, SpeedStep, and HyperThread extensions were disabled, to get stable results. We used the Intel SGX SDK 1.7 for Linux and set the stack size at 4 KB and the heap size at 1 MB.

As shown in Fig. 6, time spent in the authentication procedure with the UniSGX module increases according to the shadow file size, as each call needs to load and unseal the file to perform the credential checking. Our results show that the use of an authentication service makes the time spent in this procedure stable, regardless of the file's size, when we consider a high number of authentication requests.

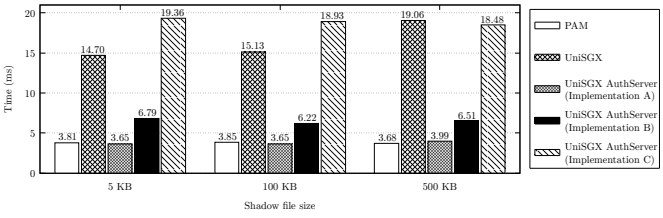


Fig. 6. Average time spent in the authentication procedure.

In the first authentication request, due the enclave creation and the load of the sealed shadow file, time spent in the authentication procedure is high and depends on the file size. However, in the following requests, the enclave is already created and the shadow file is loaded, so the authentication is faster, as shown in Fig. 7. Thus, there is a considerable reduction of the average time spent in the procedure.

The results also show that implementation A has a minor overhead, achieving an average time lower than PAM Auth module, and the time spent in the authentication procedure does not increase with the shadow file size, unlike in UniSGX.

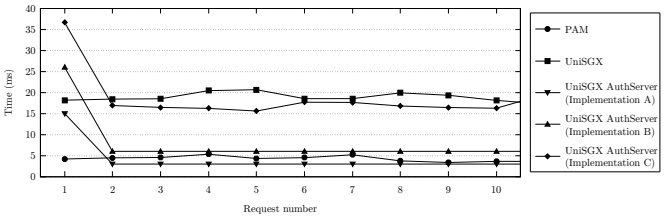


Fig. 7. Time spent on the first 10 requests using a 500 KB shadow file.

Implementation B shows an overhead due the key agreement step, presenting a 75% increase in average time when compared to PAM Auth module, but still has a small impact when compared to the original UniSGX implementation.

Implementation C, bringing the attestation procedure, shows a significant overhead due to the creation of the client enclave in each request, the encryption and decryption of the messages, and the attestation itself. This implementation is also the most secure of the three, with all client-server communication being performed between enclaves, ensuring that the untrusted component of the server will not have access to received and sent data. Despite the overhead, time spent in the authentication procedure is also constant, which makes this approach promising for environments with a large number of users, which results in a large shadow file. In addition, according to [26], time spent in the authentication procedure by this approach is not enough to be perceived by the end user.

We also evaluated the throughput of the implementations. For that, we launched 50 processes, each of them sending 20 authentication requests, and measured the time spent to satisfy all requests. Results are presented in Fig. 8, showing the number of requests that each implementation is able to satisfy per second.

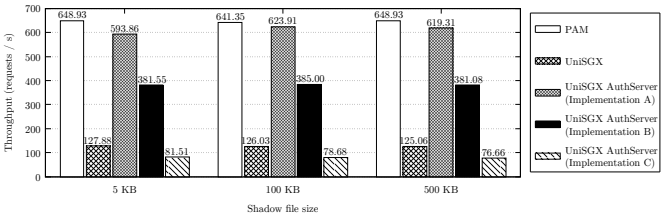


Fig. 8. Throughput of each implementation.

Results show that the enclave shared approach is reliable in replying concurrent requests, with implementation A having a highest throughput, very close to PAM Auth module. Implementation B has a good throughput, being able to answer around 380 requests per second, approximately 3 times compared to UniSGX module. Implementation C has the lowest throughput, due to the restrictions imposed by the attestation procedure, but still is able to reply 76 requests per second, in the worst case.

In addition, it is possible to emphasize the efficient use of the limited PRM by the server process, with only one enclave being necessary to manage and check the credentials for all requests, even when the requests are sent simultaneously.

### B. Security Assessment

Our threat model considers that the adversary has access to the local operating system as root, thus she can capture the sealed shadow file and mount an offline attack to crack hashes in a high performance environment. She can also perform a memory dump or a DMA operation to extract the credentials while they are being processed by the enclave. The adversary can also listen to the message bus to obtain the users' credentials.

The sealed shadow file is encrypted by a 128 bit AES-GCM algorithm, and data contained in the PRM are encrypted by a 128 bit AES-CTR algorithm. Even in a high performance environment, the adversary will spend a considerable time to crack these algorithms and get access to the hashes.

By listening the message bus, the adversary can get all data travelling on it. None of the implementations leak in the message bus any plaintext sensitive data about the users' credentials. The server sends back only the result of checking the credentials provided by the client. Information leakage among clients is also prevented by sequential handling of requests, and the enclave handles only one request at a time, with the others being queued.

Implementations A and B encrypt the data put in the message bus using a 128 bit AES-CTR algorithm. If the attacker can replace the data in the message bus, he can modify the encrypted data in order to mislead the authentication process. As the encrypted data in the bus is not authenticated, this change will not be detected, but the decrypted data will be different from the original one, and useless. Implementation B is susceptible to active man-in-the-middle attacks, with the attacker being able to replace the keys sent in the agreement process by their own keys.

Implementation C encrypts all data in the bus using an authenticated 128 bit AES-GCM algorithm, with a key provided by the attestation process. In this implementation, as messages between attested enclaves are authenticated, any change will be detected by the message receiver and the operation will be cancelled. This implementation also mitigates man-in-the-middle attacks, as the keys sent in the agreement process are signed.

The client/server architecture implies a single point of failure: the server application. Thus, the server application must be resilient and able to recover from failures, to avoid a denial of service in the authentication procedure.

In addition, we only modify the *auth* PAM module, but the presented approach can be extended to other PAM modules. As a proof of concept, only MD5 hash function was implemented, but more robust hash functions should be added in a real-world deployment. Also, the data input is still untrusted and vulnerable to keyloggers or memory sniffing; solutions to this problem are discussed in [27]–[29].

## VIII. Conclusion

This paper presented an approach to reduce the performance impact of SGX enclaves by using enclave sharing instead of using multiple enclaves to manipulate the same data. In order to validate our enclave sharing approach, we implemented three variants of an authentication service using a client/server architecture, to protect the credential file in Linux systems and to perform the authentication procedure in a trusted environment, even when the operating system or hypervisor is untrusted.

The performance evaluation shows that the prototypes are effective and with an almost constant authentication time with respect to the credential file size, allowing the use of this approach in high demand environments. The security assessment shows that the prototypes are feasible in the presented scenarios.

Implementation B presents a practical usage of the authentication service, establishing a secure communication channel between the client and the authentication service with a key agreement. This implementation can be improved by using an authenticated encryption algorithm, like the AES-GCM. Implementation C shows a more secure approach, using the attestation procedure to establish a secure communication channel and to ensures that the client is running at the same platform. Both implementations can be also applied in a distributed environment, with minor changes.

## Acknowledgment

This research was developed in the context of the H2020 - MCTI/RNP *Secure Cloud* project. The authors also thank the UFPR and UTFPR Computer Science departments.

## References

[1] A. Greenberg, "Hackers are passing around a megaleak of 2.2 billion records," 2019, https://www.wired.com/story/collection-leak-usernames-passwords-billions/.

[2] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Tel-Aviv, Israel: ACM, 2013.

[3] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han, "OpenSGX: An open platform for SGX research," in *Proceedings of the Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, February 2016.

[4] Intel, *Intel Software Guard Extensions Developer Guide*, 2016.

[5] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Tel-Aviv, Israel: ACM, 2013.

[6] V. Samar, "Unified login with Pluggable Authentication Modules (PAM)," in *Proceedings of the 3rd ACM Conference on Computer and Communications Security*. New Delhi, India: ACM, 1996.

[7] K. Geisshirt, *Pluggable Authentication Modules: The Definitive Guide to PAM for Linux SysAdmins and C Developers*. Packt Publishing, 2007.

[8] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with intel SGX," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. Savannah, GA, USA: USENIX Association, Nov. 2016.

[9] I. Sfyrakis and T. Gross, "UniGuard: Protecting unikernels using Intel SGX," in *Proceedings of the IEEE International Conference on Cloud Engineering*. Orlando, FL, USA: IEEE, April 2018.

[10] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "SGX-LKL: Securing the host OS interface for trusted execution," *arXiv preprint arXiv:1908.11143*, 2019.

[11] C. C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, CA, USA: USENIX Association, 2017, pp. 645–658.

[12] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux applications with SGX enclaves," in *Proceedings of the Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2017.

[13] L. Richter, J. Götzfried, and T. Müller, "Isolating operating system components with Intel SGX," in *Proceedings of the 1st Workshop on System Software for Trusted Execution*. Trento, Italy: ACM, 2016.

[14] V. Karande, E. Bauman, Z. Lin, and L. Khan, "SGX-Log: Securing system logs with SGX," in *Proceedings of the Asia Conference on Computer and Communications Security*. Abu Dhabi, United Arab Emirates: ACM, 2017.

[15] R. C. R. Condé, C. A. Maziero, and N. C. Will, "Using Intel SGX to protect authentication credentials in an untrusted operating system," in *Proceedings of the IEEE Symposium on Computers and Communications*. Natal, RN, Brazil: IEEE, 2018.

[16] B. A. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: Functional encryption using Intel SGX," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Dallas, Texas, USA: ACM, 2017.

[17] A. T. Gjerdrum, R. Pettersen, H. D. Johansen, and D. Johansen, "Performance of trusted computing in cloud infrastructures with Intel SGX," in *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. Porto, Portugal: SCITEPRESS, 2017.

[18] C. Zhao, D. Saifuding, H. Tian, Y. Zhang, and C. Xing, "On the performance of Intel SGX," in *Proceedings of the 13th Web Information Systems and Applications Conference*. Wuhan, China: IEEE, sep 2016.

[19] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "HardIDX: Practical and secure index with SGX," in *Proceedings of the 31th Data and Applications Security and Privacy*. Philadelphia, PA, USA: Springer, 2017.

[20] Y. Shen, Y. Chen, K. Chen, H. Tian, and S. Yan, "To isolate, or to share?: That is a question for Intel SGX," in *9th Asia-Pacific Workshop on Systems*. Jeju Island, Republic of Korea: ACM, 2018.

[21] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, A System of Patterns*, ser. Pattern-Oriented Software Architecture. Wiley, 2013, vol. 1.

[22] D. Li, R. Lin, L. Tang, H. Liu, and Y. Tang, "SGXPool: Improving the performance of enclave creation in the cloud," *Transactions on Emerging Telecommunications Technologies*, 2019.

[23] H. Dang and E.-C. Chang, "Autonomous membership service for enclave applications," *arXiv preprint arXiv:1905.06460*, 2019.

[24] freedesktop.org, "D-Bus," December 2018. [Online]. Available: https://www.freedesktop.org/wiki/Software/dbus/

[25] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *Proceedings of the International Workshop on Public Key Cryptography*. New York, NY, USA: Springer, 2006, pp. 207–228.

[26] C. Kohrs, N. Angenstein, and A. Brechmann, "Delays in human-computer interaction and their effects on brain activity," *PLOS ONE*, vol. 11, no. 1, 2016.

[27] S. Weiser and M. Werner, "SGXIO: Generic trusted I/O path for Intel SGX," in *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*. Scottsdale, AZ, USA: ACM, 2017.

[28] T. Peters, R. Lal, S. Varadarajan, P. Pappachan, and D. Kotz, "BASTION-SGX: Bluetooth and architectural support for trusted I/O on SGX," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. Los Angeles, CA, USA: ACM, 2018.

[29] A. Dhar, E. Puddu, K. Kostiainen, and S. Capkun, "Proximitee: Hardened SGX attestation and trusted path through proximity verification," Cryptology ePrint Archive, Report 2018/902, 2018.