I Can't Escape Myself: Cloud Inter-Processor Attestation and Sealing using Intel SGX

Daniel Andrade João Nuno Silva Miguel Correia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa – Lisboa, Portugal
{daniel.andrade, joao.n.silva, miguel.p.correia}@tecnico.ulisboa.pt

***Abstract*—SGX enclaves protect the code and data within from untrusted software, but do not retain their state when destroyed. Sealing allows preserving this data for future use by storing it outside the enclave boundary: the data is encrypted with a fresh secret key, and this key is bound to the processor that sealed the data and, either to the enclave measurement, or the public key of the enclave author. However, in a cloud environment, customers do not choose in which processor their code executes: the enclave that seals some data (for backup or future use) may be destroyed and later instantiated on a different processor or migrated to another processor. In those cases, the new processor would not be able to unseal the data, since the secret key is bound to the sealing processor.**

This paper presents *Inter-Processor Attestation and Sealing* (IPAS), a novel sealing mechanism for cloud environments and Intel SGX. In IPAS, sealed data is no longer bound to the sealing processor, but only to the enclave measurement or the public key of the enclave author, thus enabling other processors to unseal the sealed data. This is achieved without exporting the sealing secret key outside the enclave and without trusting third parties.

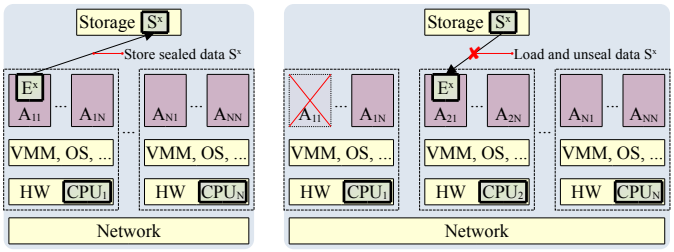
***Index Terms*—cloud computing, Intel SGX, mutual attestation, inter-processor sealing, TEE, enclave**

I. Introduction

As the use of *cloud computing* increases [1], [2], so does the need for better protection for the customer code and data that are handled by cloud providers [3], [4], [5]. Processor manufacturers have come up with hardware-based solutions such as Intel Software Guard Extensions (SGX) [6], AMD Secure Encrypted Virtualization (SEV) [7], and ARM TrustZone [8] to support Trusted Execution Environments (TEEs).

SGX allows user-space processes to create protected regions of memory called *enclaves*. Code and data within an enclave are isolated from other enclaves, and their confidentiality and integrity are protected from unprivileged and privileged software, including the operating system and hypervisor. This is of particular importance for cloud applications [9], [10], [11], [12], [13], as customers do not control the hosting infrastructure.

Enclaves use a mechanism called *attestation* [14] to prove to other enclave or non-enclave applications and services that a set of information is true. Examples of such information are the enclave identity and whether it is running on a legitimate TEE. Enclaves do not retain their state when destroyed, but enclave data can be saved for future instances of the enclave by storing it outside the boundary of the enclave, in persistent memory, using a mechanism called *sealing* [14]. Sealing encrypts enclave data with a secret key, called the *sealing key*, which is cryptographically bound to either the enclave identity or the enclave author, and to the processor.



(a) E^x in A_{11} seals enclave data as S^x.

(b) E^x in A_{21} cannot unseal the previously sealed enclave data.

Fig. 1. Scenario that shows that in SGX sealing the unsealing processor must be the same as the sealing processor. E^x in A_{21} is unable to unseal S^x previously sealed by E^x in A_{11} because the sealing and unsealing processors are different.

A cloud provider usually has many enclave instances running in its infrastructure. When an enclave instance is destroyed, there is no guarantee that a future instance of that same enclave will run on the same CPU. In Figure 1a, the enclave instance E^x (where x is the enclave identity, see § II-A), which is a component of the application A_{11}, seals enclave data into a blob S^x, and stores S^x in some shared persistent storage. In Figure 1b, which represents some point in the future of the same system, A_{11} that is running on CPU 1 is destroyed and re-instantiated as A_{21} running on CPU 2. The enclave instance E^x in A_{21} loads S^x from the shared persistent storage and tries to unseal it, but the unsealing operation fails because the sealing key that sealed S^x is bound to CPU 1. The same issue could happen when, instead of being destroyed, the enclave instance that sealed some enclave data is migrated from the sealing processor to a different processor (e.g., from CPU 1 to CPU 2); or when the sealing processor is decommissioned.

Inter-Processor Attestation and Sealing (IPAS): We solve this limitation of *SGX sealing* by removing the binding between the sealing key and the processor. Our solution requires the ability to mutually attest enclave instances running on different processors; since this is not available in the Intel SGX SDK, we built our own mutual attestation mechanism.

The IPAS scheme is composed of three core protocols:

- the *Inter-Processor Attestation* (IPA) protocol that handles the mutual attestation of enclave instances running

on different processors;
- the *Inter-Processor Sealing* (IPS) protocol that handles sealing of enclave data in such a way that it is bound to the enclave identity or the enclave author, but not to the sealing processor;
- the *Inter-Processor Unsealing* (IPU) protocol that unseals data sealed with IPS.

Inter-Processor Attestation uses enclave quotes as a transportation mechanism to exchange the enclaves' public keys safely. An *enclave quote* is a credential that is signed by a platform-specific private key, and that reflects the state of the enclave and the platform where the enclave instance is running (§ II-A). Each quote has a field to accommodate up to 64 bytes of user data [15]. In IPA, a hash of the public keys of the enclaves is stored in the first 32 bytes of this field, so these keys are implicitly endorsed by the *Intel Attestation Service* (IAS) [16] when it issues a (signed) *attestation report* based on this enclave quote. Each enclave knows that its peer is authentic and that the correct public keys have been exchanged by validating the attestation reports.

Inter-Processor Sealing uses an untrusted service, running on a platform operated by the Cloud Provider, as a support to seal enclave data. The application sends to this service the shared library (i.e., the `enclave.signed.so`) used to create its application enclave; the service then uses the shared library to create a new enclave instance, which we call service enclave, with the same measurement as the application enclave. This results in two instances of the same enclave, one located in the application and the other located in the service. After a successful mutual attestation between the application enclave and corresponding service enclave, using the IPA protocol, the application enclave sends the IPAS sealing key, with which it encrypts its enclave data, to the service enclave. The service enclave seals the IPAS sealing key to itself using SGX sealing and returns the sealed version of the IPAS sealing key to the application enclave for appending to any sealed data blobs the application enclave creates. When the same application enclave is created on another processor, it asks the untrusted service to create an instance of the service enclave to SGX-unseal the IPAS sealing key. The idea is that the service runs exactly the same enclave as the application, that is, the application enclave is trusting its own code, not a third party. In addition, the service enclave can always SGX-unseal the IPAS sealing keys because the Cloud Provider can ensure that the untrusted service always runs on the same processor and is not migrated.

Alternative designs: (1) An alternative design to IPAS would be to exchange the sealing key directly between the sealing and unsealing instances using a key exchange protocol. However, this scheme would only work when instances are online simultaneous, which is a limitation that IPAS solves. (2) Another alternative would be to use a trusted backend service that verifies the identities of the sealing and unsealing enclaves and facilitates the exchange of the sealing secret key [15]. The Service Provider could run the backend trusted service in its premises, but this would require acquiring and maintaining the necessary infrastructure, and ensuring its availability and security. Alternatively, the Cloud Provider could run it, but the Service Provider would have to trust the Cloud Provider. The Trusted Computing Base (TCB) would increase in both cases. IPAS does not trust third parties as the support service IPAS uses is outside the TCB.

Applications: We believe that most cloud-based applications that use enclaves can benefit from IPAS. These applications follow a similar pattern: (i) a company processes confidential data in a cloud service that provides SGX enclaves (e.g., Amazon AWS, Microsoft Windows Azure); (ii) that confidential data or cryptographic material used to protect it is kept in enclaves to prevent disclosure; (iii) IPAS is used to seal it in a way that allows retrieving it in another CPU, for backup purposes or for switching off and later switching on the data processing infrastructure; (iv) the sealed data is stored in some data store; (v) later the data is retrieved from storage and IPAS is used to unseal the data in some enclave, typically in an other CPU (but it can be the same).

Examples are: (i) a bank sends customer data to the cloud to do analytics, but that data is business critical[1]; (ii) a company stores genetic data in the cloud to provide services that require processing, but that data is protected by legislation like US HIPAA[2]; (iii) a recommendation system run in a cloud needs its data protected since personal data is protected by EU's GDPR[3]; (iv) Blockchain wallets kept in the cloud.

Experiments: We implemented and evaluated experimentally the performance of IPAS. The IPA protocol takes less than 4 seconds of CPU time to complete mutual attestation between two enclaves, a value that excludes network communication and IAS times, which are beyond our control. The enclaves run the protocol only once after instantiated, so we find that this is an acceptable execution time for mutual attestation. The IPS protocol and its counterpart, IPU, take less than 4 seconds of CPU time to obtain the IPAS sealing key. Data sealing itself depends on the size of the plaintext and takes roughly the same time as the original SGX sealing implementation. The source code of our prototype and the formal models and proofs are available online.[4]

Contributions: Our main contributions are the following. First, a new architecture and protocols enabling inter-processor mutual attestation of enclave instances executing on different processors and inter-processor sealing of enclave data, particularly suited for cloud environments. Second, a formal model of our protocols. Finally, a prototype of the IPAS libraries and their experimental evaluation.

Outline: Section II presents background on Intel SGX and introduces the functions used in our protocols. Section III explains the IPAS architecture and protocols. Section IV presents our formal model and proofs. Section V discusses our prototype and Section VI its evaluation. Section VII discusses related work and Section VIII concludes this paper.

[1]https://eperi.com/success-stories/intel-sgx/
[2]https://genecrypt.io/
[3]https://github.com/Ruide/PrivateLearn
[4]https://github.com/andrade/ipas

II. Preliminaries

A. Intel SGX

SGX provides a set of instructions that enable user-level code to create protected regions of memory called enclaves. Two of these instructions are EGETKEY and EREPORT. They allow one to obtain secret keys bound to the processor and to create a cryptographic report of the enclave [17], respectively. Untrusted code communicates with enclave code using *ecalls* and *ocalls*, which are, respectively, functions executed inside the enclave and outside the enclave. These functions are described in an Enclave Definition Language (EDL) file.

Each enclave contains a self-signed certificate from the author of the enclave with multiple fields, including enclave identity, enclave author, and product identity [15]. The enclave identity is a hash of the code and initial data of the enclave during its instantiation; it is stored in the MRENCLAVE register by the CPU and compared to the corresponding field in the self-signed certificate. The enclave author is a hash of the public key of the enclave author and is stored in the MRSIGNER register. The product identity is assigned by the enclave author to segment enclaves with the same MRSIGNER. These two measurements, MRENCLAVE and MRSIGNER, identify enclaves during attestation and sealing. SGX supports two basic forms of attestation: local and remote [14].

1) SGX Local Attestation: In local attestation, an enclave A attests itself to another enclave B using a report authenticated by a symmetric key, the report key, accessible only to the enclaves involved in the process.

Enclave A obtains a report by invoking EREPORT using the MRENCLAVE of the target enclave as input. This report contains information about A including its MRENCLAVE and MRSIGNER, and is authenticated by a Message Authentication Code (MAC). A sends the report to B, that extracts the report key by invoking the instruction EGETKEY and verifies the MAC of the report. If the MAC computed by B and the MAC on the report match, it means that both source and target enclaves are running on the same platform. Enclave B then validates the report's contents, namely MRENCLAVE and MRSIGNER. B can also attest itself to A in a similar fashion.

2) SGX Remote Attestation: In remote attestation, an enclave A attests itself to a challenger C using asymmetric cryptography. C may or may not use an enclave. Remote attestation uses a local architectural enclave called the Quoting Enclave [14] and a remote service, the Intel Attestation Service (IAS) [16].

Enclave A attests itself to the Quoting Enclave using the local attestation mechanism. Upon a successful local attestation, the Quoting Enclave converts the local report into a signed quote that can be verified by Intel. This quote is sent to C, which forwards the quote to IAS to be validated. IAS verifies the signature and validates the fields on the quote and replies to C with a signed attestation report. C verifies the IAS signature of the attestation report and checks whether IAS considers A to be legitimate; and then decides whether to proceed its communication with A based on the attestation report's information.

Enclaves A and C can establish session keys, at the end of remote attestation, for secure communication. A uses the trusted key exchange functions of the Intel SGX SDK [17, p. 144] to obtain its session keys MK and SK, but C derives these keys manually [18] using the function in Line 5 of Algorithm 1.

Algorithm 1 Derive session key with label s

```
1: function KDK(d_a, Q_b)                          ▷ Key Derivation Key
2:     k_ab ← d_a · Q_b                            ▷ ECDH
3:     return AES-128-CMAC(k_ab, 0^128)
4: end function

5: function DERIVEKEY(s, d_a, Q_b)                 ▷ private key, d_a
6:     m ← 0x01‖s‖0x00‖0x80‖0x00                   ▷ public key, Q_b
7:     k ← KDK(d_a, Q_b)
8:     return AES-128-CMAC(m, k)
9: end function
```

3) SGX Intra-Platform Sealing: SGX provides confidentiality and integrity to data within the boundary of an enclave. This data is lost when the enclave is destroyed. Sealing is a mechanism where data is encrypted (using authenticated encryption to ensure both confidentiality and integrity) and bound to a processor and to either the enclave measurement or the public key of the enclave author. Sealed data can then be saved to persistent memory such as a hard disk drive.

Data sealed to the enclave identity can be unsealed only by another enclave with the exact same MRENCLAVE, while data sealed to the enclave author can be unsealed by any enclave with the same MRSIGNER. The product identity must be the same in both cases. In addition, the sealing and unsealing processors must be the same because the sealing key derivation depends on the processor.

Sealing data to the enclave author has the benefit that it is easier to update an enclave (MRENCLAVE changes when the enclave is updated) and enables enclaves with the same enclave author to share sealed data [14]. The decision to seal data to the current enclave measurement (MRENCLAVE) or to the enclave author (MRSIGNER) is one of policy and is taken by the enclave author.

We use the term *SGX-seal* when referring to the traditional sealing mechanism of Intel SGX (§ II-A) and the term *IPAS-seal* when referring to the novel sealing mechanism introduced in this paper (§ III-E).

B. Functions and Assumptions

We now define several functions used by the IPAS protocols.

There is an IND-CPA *symmetric encryption* scheme modeled with functions $\text{senc}(m, k)$ that takes a message m and a secret key k as inputs and outputs a ciphertext; and $\text{sdec}(c, k)$ that takes a ciphertext c and a secret key k as inputs and outputs a plaintext.

There is an IND-CPA and IND-CTXT *authenticated encryption with associated data* scheme modeled with functions $\text{enc}(m, a, n, k)$ and $\text{dec}(c, t, a, n, k)$. The first function takes a

message m, associated data a, a nonce n, and a secret key k as inputs, and outputs a ciphertext c and a tag t. The second function takes a ciphertext c, a tag t, associated data a, a nonce n and a secret key k as inputs and outputs a plaintext. This scheme provides confidentiality, integrity, and authenticity for encrypted plaintext, as well as integrity and authenticity for some associated data that is not encrypted [19].

There is a function create(e) that creates and starts an instance of enclave image e.

There is a function seal(m) that SGX-seals the message m to the processor that executes the function and a function unseal(s) that SGX-unseals the sealed data blob s using the processor executing the function. The unseal() function returns an error when the processor is unable to unseal the data.

There is a function load(b) that loads data b from persistent memory into the enclave, and a function store(b) that stores data b from the enclave to persistent memory.

III. Inter-Processor Attestation and Sealing

Intel SGX provides mechanisms for local and remote attestation, and intra-platform sealing. IPAS (Inter-Processor Attestation and Sealing) provides mechanisms for the *mutual* attestation of enclaves running on the same or on different processors, and *inter*-platform sealing, which are not present in SGX. This section begins with the threat model and architecture of IPAS and then details the IPA (Inter-Processor Attestation) protocol and the IPS (Inter-Processor Sealing) and IPU (Inter-Processor Unsealing) protocols. As both SGX and IPAS seal, unseal, and attest enclaves, we use terms like *IPAS-seal* and *SGX-seal* to clarify which operation we are talking about.

A. Threat Model

The adversary controls the network and all unprivileged and privileged software on the platforms. Side-channel attacks and denial-of-service attacks are beyond the scope of this work.

The TCB consists of processor package and the enclaves, which are implemented and working correctly. The private and secret keys of the processors are not compromised.

B. Architecture

Figure 2 shows the key components of the system. The architecture is composed of an application ⓪ containing an enclave (2,4) that wants to seal its data or mutually attest with another enclave; and three support services, the *Cloud Sealing Service* (CSS) ⑥, the *Remote Attestation Proxy* (RAP) ⑦, and IAS ⑧. IAS is a service in operation provided by Intel; CSS and RAP are introduced by us as part of IPAS. Only one application is considered, but in production there would be many, each with its own enclave.

Applications (called enclave applications ⓪ to explicitly denote that they contain an enclave) are owned by Service Providers and run within data centers of Cloud Providers. Service providers are stakeholders, for example, a bank or a healthcare provider, who run their enclave applications in a cloud environment. Each application is partitioned into two logical components: the trusted component (2,4) is the enclave (there may be more than one in the same application) and the untrusted component (1,3) is the remaining code of the application. Application programmers develop the untrusted ① and trusted ② application code, and IPAS libraries provide the untrusted ③ and trusted ④ IPAS code to programmers.

Enclaves (2,4) are part of applications and services. They use the IPAS libraries to have access to IPAS mutual attestation and IPAS sealing and unsealing. There are two main use cases for the IPAS protocols:

1) one where the goal is sealing (or unsealing) data;
2) and the other when the goal is the mutual attestation between two enclave instances.

The first use case is when an enclave instance wants to IPAS-seal enclave data (or IPAS-unseal previously sealed data). The enclave instance first triggers an execution of the IPA protocol between its hosting application and CSS, and then runs the IPS protocol (or IPU protocol) between the enclave instance on the application and the enclave instance in CSS. This is useful whenever the enclave wants to seal its data for later use, which can happen at any point during its execution, and particularly before destroying the enclave and terminating the application. The second use case is where two instances of the same enclave, located on the same or on different platforms, want to mutually attest each other to ensure that both are legitimate and up-to-date. In this case, the IPA protocol is run between both instances without need to communicate with CSS.

1) Cloud Sealing Service (CSS): CSS ⑥ is a key component of our solution. It is a service to be deployed by the Cloud Provider to help application enclaves in the processes of IPAS sealing and IPAS unsealing. CSS can always unseal data it previously sealed because the Cloud Provider can ensure that it always runs on the same processor and is not migrated. The virtual machines of client applications differ from CSS since, in a cloud environment, these are subject to migration from one (physical) server to another and are usually reinstantiated in different (physical) servers when restarted after a period of inactivity.

CSS is *not trusted*. CSS SGX-seals and SGX-unseals its clients' keying material inside enclaves. CSS supports application enclaves during IPAS sealing/unsealing by creating enclave instances, that we call *service enclaves*, with the exact enclave measurements as its clients, and then sealing the clients' IPAS sealing keys on their behalf. First, the application enclaves send their IPAS sealing key to service enclaves in CSS. Then, these service enclaves SGX-seal the IPAS sealing key and return the result to the client to be stored in persistent memory. These actions are done after mutual attestation between the application and service enclaves. In a future instantiation, the application enclave can request CSS to create a service enclave to SGX-unseal the application enclave's IPAS sealing key. Since each IPAS sealing key is handled by CSS inside a service enclave with the same MRENCLAVE and MRSIGNER as the client's application enclave, vulnerabilities in one service enclave, that could

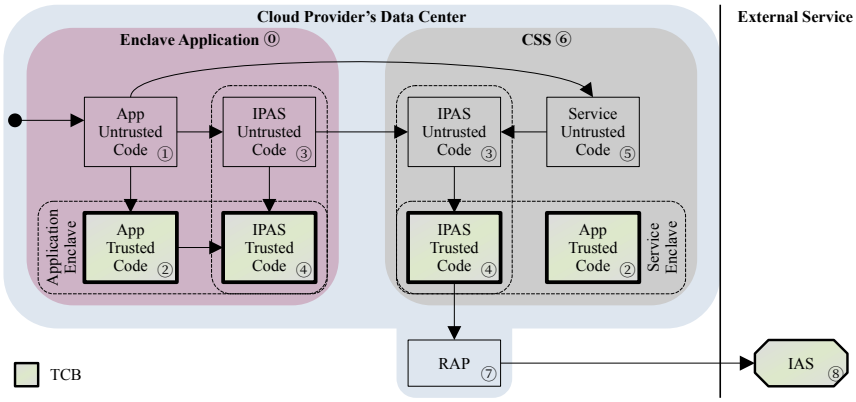


Fig. 2. The IPAS system architecture. The application enclave and the service enclave instances are created from the same trusted code, that is, the application trusted code and the IPAS trusted code. The application untrusted code first creates its own enclave instance (called application enclave) and then sends the necessary modules to the CSS untrusted code which also creates its own enclave instance (called service enclave).

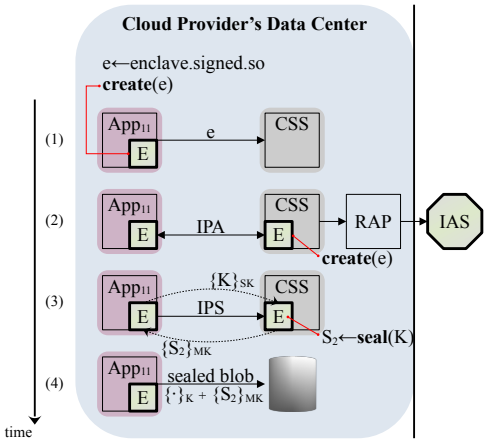


Fig. 3. High-level overview of the IPA and IPS protocols.

potentially expose the IPAS sealing key, do not affect other service enclaves.

CSS does not run enclaves when it starts or when it is idle, only untrusted code (⑤). At runtime, clients connect to CSS and send their shared libraries with enclave code (②,④), and related untrusted code (③), over the network; then the untrusted code of CSS (⑤) uses the API of IPAS to create service enclaves, from these shared libraries, to execute the IPA, IPS, and IPU protocols. CSS communicates with RAP during IPA to obtain attestation reports signed by IAS.

Each library received by CSS is composed of both application (②) and IPAS (④) trusted code. This trusted code may invoke untrusted functions, defined in an EDL file, from within the enclave, which therefore must also be available in CSS. For this reason, applications send to CSS not only the signed shared library representing the enclave but also a shared library containing the untrusted functions (③) defined in the EDL files of the application and IPAS.

In this paper, we use the term *application enclave* when referring to an enclave that is part of a Service Provider's application, and *service enclave* when referring to an enclave that is part of CSS.

2) Remote Attestation Proxy (RAP): RAP ⑦ is deployed by the Cloud Provider to support enclaves during the execution of the IPA protocol by forwarding requests to and from IAS. Clients do not communicate directly with IAS because that would requires clients to authenticate using a subscription key obtained from Intel. Handling this subscription key in each client would complicate deployment, first because the subscription key must be kept secret, and second because of subscription key rotation. Having only one entity handling the subscription key simplifies both tasks. The confidentiality of the subscription key must also be protected in RAP. We do not provide one solution, as there are a few options: using an enclave and sealing the key before storing it outside the enclave boundary; or using a hardware security module.

RAP allows the attestation of enclaves, including those created by the CSS service. These enclaves use the functions `GetSigRL` and `GetReport` to retrieve the signature revocation list for a specific Intel EPID Group ID [20] (see next paragraph) and obtain an attestation report for a specific quote, respectively. In both cases, the arguments sent by clients to RAP are forwarded to IAS, with which RAP can communicate using its subscription key, registered with Intel. The results returned by IAS are forwarded by RAP to the clients.

3) Intel Attestation Service (IAS): IAS ⑧ is provided and controlled by Intel [16]. Enclaves communicate with IAS, through RAP, during the IPA protocol. For using IAS in production, an organization has to subscribe the Intel SGX attestation service. IPAS is based on Intel's Enhanced Privacy ID (EPID) attestation scheme, which allows signing data without allowing linking two signatures for privacy reasons. Each EPID signer belongs to a *group* (or EPID group) and signature verifiers use the group public key to verify the signature, independently of the entity that signed the data.

C. Overview of the IPAS Protocols

IPAS is composed of three protocols: IPA, IPS, and IPU. Together, they enable mutual attestation of enclaves executing on different processors, as well as sealing and unsealing of data bound to an enclave or signer but not bound to the processor, unlike what happens with the equivalent operations offered by the Intel SGX SDK. This is achieved without exposing the sealing key outside the enclaves and without trusting third parties.

Mutual attestation is achieved by running the IPA protocol (§ III-D) between any two enclave instances with the same MRSIGNER and, optionally, the same MRENCLAVE, depending on how the enclaves were configured. Sealing and unsealing are used by an application enclave that needs to seal or unseal its data, and are carried out by running the IPS protocol (§ III-E) or the IPU protocol (§ III-F) between that application enclave on the client and a service enclave in CSS, after a successful execution of the IPA protocol. IPAS

unsealing works even when the original sealing enclave and sealing processor are unavailable.

The intuition behind IPAS sealing and unsealing is creating an enclave in CSS with the same MRENCLAVE and MRSIGNER as the enclave running on the client, that is, the application enclave and the service enclave are two instances of the same enclave running on different processors; then, this service enclave is used to SGX-seal the secret key with which the application enclave IPAS-seals its data. CSS is an always-on service similar to IAS but, unlike IAS, this service does not need to be trusted because the enclaves created (service enclaves) are attested by, and have the same measurements as, the application enclaves.

Figure 3 presents a high-level overview of the system. The figure also presents the process of mutual attestation between two enclave instances, followed by IPAS sealing. There are four steps:

1) The application creates its enclave instance, which we call *application enclave*, from an `enclave.signed.so` shared object, then sends this shared object to CSS. At this point, CSS still does not have a running enclave instance;
2) CSS creates the corresponding enclave instance, which we call *service enclave*, from the shared object it received from the client. Then, the application enclave and the service enclave mutually attest using the IPA protocol, which results in both parties sharing session keys (MK and SK);
3) The application enclave randomly generates a secret key, K, to use as the IPAS sealing key and sends it to the service enclave. K is encrypted with the session key SK in such a way as to protect the confidentiality and integrity of K while in transit. The service enclave decrypts K using SK and SGX-seals K using the traditional SGX sealing mechanism. Then, the service enclave sends SGX-sealed K, integrity-protected with MK, to the application enclave;
4) The application enclave seals its data using the IPAS sealing key it previously generated (i.e., K), appends the service-enclave-SGX-sealed data blob (which contains K) to its encrypted data, and saves both application-enclave-encrypted data and service-enclave-SGX-sealed data to persistent memory.

D. IPA Protocol

The IPA protocol allows mutual attestation between two enclaves, an initiator A and a responder B, running on the same or on different platforms. These two enclaves use Intel EPID attestation [16] during the protocol, with B being responsible for sending all requests, from both A and B, to RAP, which in turn forwards these requests to IAS.

Successful execution of the IPA protocol is a precondition for the IPS protocol (§ III-E) and the IPU protocol (§ III-F). In these two cases, the IPA protocol mutually attests A and B which would be the application enclave and the service enclave, respectively. However, IPA can be used standalone, i.e., the use of IPA does not imply the subsequent use of the IPS or IPU protocols. The IPA protocol is generic enough that it can be used, without CSS, for any two enclaves that have the same enclave identity or the same enclave author to mutually attest. For example, a healthcare provider could use the IPA protocol to mutually attest two instances of its application, to ensure that both instances are legitimate and up-to-date, before transferring data between them using a secure communication channel.

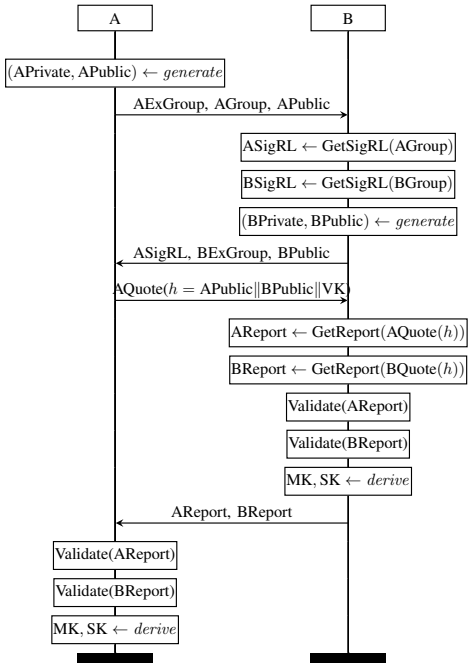


Fig. 4. Inter-Processor Attestation (IPA) protocol.

Figure 4 shows an execution of the IPA protocol between the enclaves A and B. Details of the communication with IAS, through RAP, are not shown since it follows the protocol and uses the Intel API [16]. This interaction with IAS is represented in Figure 4 by the functions `GetSigRL` and `GetReport`. The protocol works as follows:

1) Enclave A generates a fresh key pair and sends the corresponding public key (APublic), as well as its Extended Group ID (AExGroup) and Group ID (AGroup) to B. The Extended Group ID and the Group ID are obtained using the SGX SDK and identify the EPID group [20] to which the attesting platform belongs.
2) Enclave B requests the signature revocation lists of both A and B (ASigRL and BSigRL) from IAS, through RAP, using the enclaves' Group IDs (AGroup and BGroup) as input. Then, B generates a key pair and sends the corresponding public key (BPublic), its Extended Group ID (BExGroup), and the signature revocation list of A (ASigRL), to A. Each EPID SigRL contains a list of revoked platforms [20]. Each quote produced in the next steps contain a proof that the corresponding platform is not on the list.
3) Enclave A obtains a quote (AQuote), containing h as user data, and sends it to B.
4) Enclave B obtains a quote (BQuote), containing h as user data, and requests attestation reports for both A and B

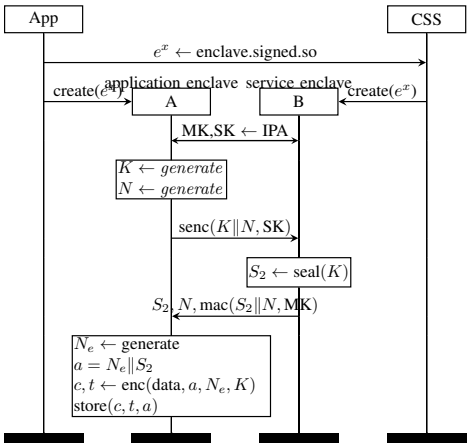


Fig. 5. Inter-Processor Sealing (IPS) protocol.

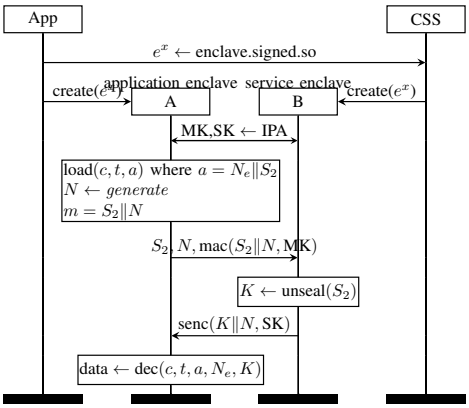


Fig. 6. Inter-Processor Unsealing (IPU) protocol.

(AReport and BReport) from IAS, through RAP, using the enclaves' quotes (AQuote and BQuote) as input. Enclave B then validates both attestation reports (AReport and BReport), derives session keys (MK and SK), and sends the attestation reports to A.

5) Similarly, A validates both attestation reports and derives session keys.

Enclaves A and B validate the attestation reports, AReport and BReport, produced and signed by IAS, by verifying the fields of each report. In particular, the fields `isvEnclaveQuoteStatus` and `isvEnclaveQuoteBody.ReportData` contain, respectively, the state of the enclave, as judged by IAS, and the user data created by the enclaves at the beginning of the attestation process. In the first field, we expect to find `OK` which indicates that the enclave and the processor are legitimate and up-to-date. In the second field, we expect to find the same h that was sent as part of the quotes of A and B. Each attestation report is signed by a report signing key, owned by IAS, which is verified using the corresponding public key hard-coded in the enclaves. The MRENCLAVE field is compared in both attestation reports and should be the same; otherwise, mutual attestation fails even when all else is in an acceptable state. This can be relaxed by comparing the MRSIGNER field instead of MRENCLAVE, and depends on how the mutual attestation of enclaves is configured.

The user data h is obtained by computing the SHA-256 hash of the concatenation of the public keys of A and B with the verification key VK. The derivation of VK, and the session keys MK and SK, follows the function in Algorithm 1, Line 5. The input s is "VK", "MK" or "SK" according to the desired key, and the inputs d_a and Q_b are the private key of the entity that derives the key and the public key of its peer, respectively.

E. IPS Protocol

The IPS protocol allows an application enclave to seal data in such a way that another instance of the same enclave, running on the same or a different processor, is able to unseal the sealed data. The unsealing instance can unseal the sealed data even when the sealing instance is no longer available, for example, because the enclave has been destroyed.

Figure 5 shows the application enclave A sealing application data. For this purpose, A takes advantage of the service enclave B created by CSS. The protocol assumes as a precondition that the IPA protocol was previously successfully executed between A and B, resulting in session keys MK and SK.

Enclave A generates a secret key, K, to use as IPAS sealing key, and a nonce, N, and sends both (K, N) encrypted with session key SK to B. Enclave B SGX-seals K, binding it to its own processor, and returns the sealed secret key S_2 and the nonce N to A. A verifies if N is correct, generates a second nonce, N_e, encrypts its data using K and N_e, and saves the ciphertext and additional data outside the enclave. The sealed secret key, S_2, and the data encryption nonce, K_e, are part of the encryption algorithm as additional data.

The IPS protocol has to run only once per enclave instantiation since K can be reused by generating a different data encryption nonce, K_e, for each sealed data blob.

F. IPU Protocol

The IPU protocol unseals data previously sealed using the IPS protocol, by another enclave instance, with either the same MRENCLAVE or the same MRSIGNER, running on the same or on a different processor. The original sealing enclave and sealing processor are not required by the protocol, which means that unsealing works even when the sealer is unavailable, for example, due to the sealing enclave having been destroyed or the sealing processor having been decommissioned.

Figure 6 shows the unsealing of data by the application enclave A with the support of the service enclave B created by CSS. The protocol assumes, as a precondition, that the IPA protocol was previously successfully executed between A and B, returning the session keys MK and SK.

Enclave A loads the IPAS sealed data bundle into the enclave, extracts S_2 from AD, generates a nonce, N, and sends both S_2 and N, authenticated with MK, to B. Enclave B SGX unseals S_2, which it has previously sealed, and returns the resulting secret key, K, encrypted with the SK session key, to

A. Enclave A decrypts $K\|N$ using SK, verifies that the nonce N is correct, and decrypts its client data using K and N_e.

IV. Security Analysis

We analyze IPAS security using *Tamarin*, a recent security protocol verification tool that is being much used to prove security protocols [21]. In Tamarin a Dolev-Yao adversary controls the communication network. We modeled the protocols following the three sequence diagrams in Section III.

1) Analysis of the IPA Protocol: The formal model of the IPA protocol follows the sequence diagram in Figure 4 but with modifications to help with the proof. The AExGroup, AGroup, ASigRL and BExGroup are not exchanged in the model, since these are unnecessary. The Extended Group ID is always zero [22]. The Group ID is used to obtain the Signature Revocation List, which in turn is used to obtain the IAS attestation report. However, since we do not model the revocation of SGX processors, we can obtain the attestation report in our model without needing the corresponding Signature Revocation List.

We use the following lemma to prove the secrecy of the session keys MK and SK:

```
All k #i. Secret(k) @i ==>
(not (Ex #j. K(k) @j)) | (Ex X #j. Leak_private_DH(X) @j)
```

The lemma states that whenever the action `Secret(k)` occurs at timepoint `i`, either `k` was leaked or the adversary does not know `k`. The action is applied to MK and SK (`Secret(MK)` and `Secret(SK)`) at the end of the IPA protocol, both in the initiator and responder, which implies that these two secret keys were not compromised at such a time when the protocol ends. The proof takes 434 steps in Tamarin.

2) Analysis of the IPS and IPU Protocols: The formal models of the IPS and the IPU protocols follow the sequence diagrams in Figure 5 and Figure 6, respectively. These two protocols are executed after a successful execution of the IPA protocol, so in their models we simply assume that the result of running IPA is available, i.e., keys MK and SK.

In IPS the sealed data is stored outside the enclave boundary, and in IPU the sealed data is loaded from outside the enclave boundary. We model this by sending the data to what Tamarin designates as *untrusted network*, a component where the adversary can eavesdrop and modify the data, at the end of the formal sealing model, and by receiving the data from the untrusted network at the beginning of the formal unsealing model. Since the adversary controls the communication network, we are in essence giving the adversary the data stored outside the enclave boundary.

For these two models, we use the following lemma to prove the secrecy of the IPAS sealing key, i.e., the secret key, K, which encrypts and decrypts application enclave data:

```
All k #i #j. Secret(k) @i & K(k) @j ==> F
```

The lemma states that the adversary does not know `k` whenever the action `Secret(k)` occurs at timepoint `i`. The action `Secret(k)` is placed at the end of the IPS and IPU protocols, both in the initiator and responder, implying that the IPAS sealing key has not been compromised during the execution of the protocols. The secrecy proof for sealing takes 9 steps in Tamarin and for unsealing 17 steps.

V. Implementation

The implementation is composed of IPAS libraries, CSS and RAP. IPAS libraries and CSS are implemented in C and RAP is implemented in Rust. Recall from Section III-A that RAP and CSS are not trusted.

1) IPAS Libraries: IPAS is composed of an attestation library for the IPA protocol and a sealing library for the IPS and IPU protocols. These libraries are linked into the application enclave along with the other Intel SGX SDK libraries.

2) CSS: In client applications, the enclave code is usually available from the start, so the trusted code is dynamically linked using a signed shared library and the untrusted code is statically linked. In CSS, however, enclave code is unavailable when CSS is started because there are many different purposes for client enclaves, each with its own specific enclave code, and CSS needs to support them all. In this case, regular compilation of CSS results in errors where the CSS code invokes ecalls of the IPAS API. These ecalls are implemented by enclaves whose code is not yet available, because it has yet to be received from clients. To solve this, we use `dlopen` [23] to dynamically load the shared libraries received from clients and invoke ecalls using a `dlsym` [24] wrapper, which is hidden behind IPAS functions, around the ecall instead of invoking the ecall directly.

3) RAP: RAP is a proxy that forwards requests from clients to IAS and replies from IAS to clients. One such client is CSS, which uses RAP during IPA. However, the IPA protocol can be used by any two enclave instances wanting to mutually attest, it does not have to include CSS, and therefore other applications and services may also make use of RAP. In our implementation, the Service Provider ID and the Subscription Key are stored in a configuration file, but in a production environment the Subscription Key could be stored, for example, in a hardware security module. According to Intel, the Service Provider is responsible for protecting the confidentiality of the Subscription Key [16].

4) Scalability of CSS: In our prototype, CSS is a single-processor service, but this would be unviable in a production environment because it would be a single point of failure. The CSS seals clients' encryption keys to itself, and if the processor is lost, all enclaves lose access to their sealed encryption keys. The solution is to implement CSS as a multiprocessor service.

The sequence diagrams in Figures 5 and 6 show CSS as a single entity. This entity can be made up of multiple processors, in which case the encrypted key of the sealed client represented by S_2 would contain one sealed version of K for each CSS processor. This approach does not require any modifications in the client since from the caller's perspective S_2 is an opaque blob and the protocol remains the same.

Sealing in a multiprocessor architecture is more complex for CSS. During the IPA protocol, CSS would start multiple

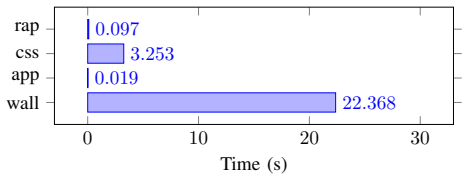


Fig. 7. IPA protocol processor execution time broken down per entity, and total wall-clock time. The processor execution time attributed to the IPA library is under 4 seconds. The remainder of the wall clock time is due to network overhead and IAS execution time which are both beyond our control.

instances of the enclave spread across N processors, with one processor working as a bridge. The caller's application enclave mutually attests with the service enclave running in the bridge and then proceeds with the IPS protocol following Figure 5 in Section III-E. Internally, the bridge enclave instance mutually attests itself with the other $N-1$ enclave instances of the service and sends them K to be sealed. Then, the bridge enclave instance itself seals K and collects the other $N-1$ versions of sealed K in a bundle S_2 and returns this bundle to the caller. The size of S_2 grows linearly with the number of processors used by CSS during the IPS protocol.

Unsealing is simple and requires only one of the N processors that make up CSS. The caller's application enclave mutually attests with the enclave instance of one of the processors of CSS and then proceeds with the IPU protocol following Figure 6 in Section III-F. Any of the N CSS processors that participated in the IPS protocol can unseal K.

VI. Evaluation

Performance was evaluated with a test application, CSS and RAP running on the same computer; and IAS running remotely on Intel-controlled servers. The test computer has an i5-7600 Intel processor (which has SGX) and 32 GB of RAM, with Ubuntu 18.04 on a solid-state drive.

Each experiment is repeated in a 1000-iteration loop, except when noted otherwise. The execution time is for the entire loop and then divided by the number of iterations when plotting the figures. Measurements are collected using `clock_gettime` [25] with `CLOCK_PROCESS_CPUTIME_ID` for measuring processor execution time and `CLOCK_MONOTONIC_RAW` for wall-clock time.

A. Mutual Attestation Protocol

Figure 7 shows the execution time of the mutual attestation protocol (§ III-D). We measured the *processor* execution time in the test application, CSS, and RAP; but not in IAS which is under the control of Intel. Additionally, we measured the wall-clock time of the full execution of the IPA protocol from beginning to end, which includes the processor execution time of the test application, CSS, and RAP as well as the networking overhead and IAS execution time. The protocol was run in a loop of 30 iterations.

The CSS processor execution time is two orders of magnitude larger than for the application and RAP. This makes sense since not only does CSS perform operations similar to the application, as seen in Figure 4, but it also handles all communication with RAP, including preparing the data in a format acceptable to IAS, serialization, and encryption. RAP mostly forwards requests and replies between CSS and IAS.

The wall-clock time of the IPA protocol is one order of magnitude larger than processor execution times of application, CSS and RAP combined. The complete execution of the protocol takes just over 22 seconds to finish, mostly due to communication with, and processing on, IAS which is beyond our control. This overhead is a reasonable trade-off for security, since the IPA protocol needs to execute only once when the two enclaves attesting themselves are started. The session keys resulting from mutual attestation can be cached until the enclaves are destroyed.

B. Sealing Protocol

We measured the processor execution time for the sealing protocol setup separately from the actual data encryption. The sealing protocol setup runs once to obtain an IPAS sealing key. This sealing key can be reused when encrypting data since IPAS generates a new nonce for each encrypted bundle. This experiment was carried out for five plaintext sizes, from 10 KiB to 100 MiB.

Figure 8 shows the processor execution time of the sealing protocol setup in the application and CSS, and their combined processor execution time. The results are similar for all plaintexts since the setup does not depend on the plaintext size.

Figure 9 compares the execution time of the enclave data sealing processor using IPAS (excluding setup) and the original SGX sealing functions. Data is encrypted with `sgx_rijndael128GCM_encrypt` and the nonce is generated with `sgx_read_rand` in both cases. However, IPAS sealing obtains the IPAS sealing key from the IPS setup protocol run with CSS and caches this sealing key, whereas SGX sealing derives the SGX sealing key with `sgx_get_key` on every run.

C. Unsealing Protocol

We measured the processor execution time for the unsealing protocol setup separately from the actual data decryption. This experiment was carried out for five different plaintext sizes from 10 KiB to 100 MiB.

Figure 10 shows the execution time of the unsealing protocol setup in the application and CSS, and their combined execution time. The results are similar for all plaintexts since the setup does not depend on the sealed data bundle size.

Figure 11 compares the processor execution time of unsealing data using IPAS (excluding setup) and the original SGX unsealing functions.

VII. Related Work

A. Enclave Attestation

IPAS uses EPID-based attestation, which requires communicating with IAS when attesting enclaves, as we describe in the background section. Intel provides an alternative to EPID called Data Center Attestation Primitives (DCAP) [26]. DCAP enables a third party, for example, a Cloud Provider, to build its own attestation infrastructure.

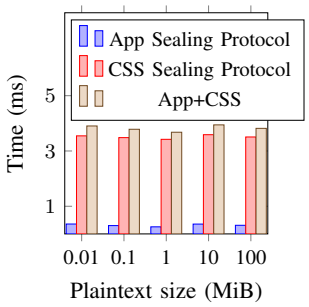


Fig. 8. IPS protocol setup time (app and CSS).

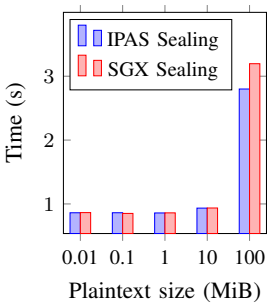


Fig. 9. Comparing IPAS with SGX sealing.

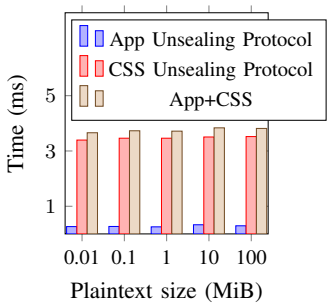


Fig. 10. IPU protocol setup time (app and CSS).

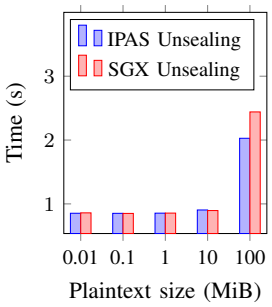


Fig. 11. Comparing IPAS with SGX unsealing.

Chen et al. [27] proposed an attestation service that improves enclave privacy during attestation, compared to EPID-based attestation, but that still uses EPID-based attestation for its backend services. Chen and Zhang [28] proposed a mutual attestation mechanism that works without trusted third parties, but does not support updates to the enclave. However, both systems still have IAS as part of their TCB. IPAS supports enclave updates because during mutual attestation the enclaves can choose between enclave-identity- or enclave-author-based mutual attestation as explained in Section III-D. Teaclave [29] solves mutual attestation by using third-party auditors that sign and publish the identities of audited enclaves. IPAS provides mutual attestation without trusting third parties.

B. Enclave Migration and Data Sealing

This work does not have the goal of presenting an enclave migration scheme, but sealing is an important component of such migration, so we analyze some related work here. eMotion extends SGX with new instructions that support the migration of running enclaves [30]. This solution requires changes in hardware and is tested using the OpenSGX open-source SGX emulator [31]. Gu et al. [32] propose a software-only solution in which each enclave runs a control thread capable of dumping the state of the enclave in the source container and restoring that state to a target container. In addition, these authors propose new instructions to support the transparent migration of enclaves. Liang et al. [33] propose a migration solution based on a trusted service that provisions migration keys to source and target client enclaves. The authors encrypt persistent state with this migration key. Nakashima and Kourai [34] propose a migration solution based on a trusted service that provisions CPU-independent keys to support data migration. Guerreiro et al. [35] propose a solution for migrating virtual machines that uses a Hardware Security Module (HSM) as a trusted entity to encrypt and decrypt enclave data on source and target machines.

Alder et al. [36] propose a software-only solution to migrate the persistent state of enclaves, including sealed data, which relies on two key features: a custom library on each enclave to be migrated; and migration enclaves, set up by the Cloud Provider, on the source and target machines.

The authors handle the sealing by generating a custom migration sealing key which is transferred, along with the client enclave's data, via the custom library and migration enclave, from source to target machine. Then the migration sealing key is stored locally, with the help of the custom library, by sealing it to the migration enclave. This solution only works for migrating sealed data when both source and target machines are live, unlike IPAS which works even when all enclave instances go offline. Furthermore, the migration sealing key becomes implicitly bound to the migration enclave, on the target machine, when it is sealed for persistent storage, which means that if this machine becomes unavailable (e.g., down for maintenance, damaged or decommissioned) then all client data, sealed by these migration sealing keys, becomes permanently inaccessible.

In comparison to IPAS, the first four migration solutions [30], [32], [33], [34] do not address inter-processor data sealing, which is the objective of IPAS. The next two [35], [36] do sealing but require trusting third-party services: an HSM set up by an administrator in one case; and a migration enclave set up by the Cloud Provider in the other. In the first case, the HSM has access to the secret keys that encrypt the clients data. In the second case, the migration enclave has access not only to the secret keys, but also to all of its clients' data. In addition, both solutions suffer from requiring an initial setup phase, where it is assumed that the system is in an adversary-free clean state. Again, IPAS does not require trusting any third party, including third-party enclaves, and does not need an initial adversary-free setup step.

VIII. Conclusion

We proposed an architecture and protocols to (i) mutually attest enclave instances executing on different processors and (ii) unseal data originally sealed in a different processor. This is achieved without requiring an initial setup step on an adversary-free clean system, without having to extend SGX with additional instructions, and without enlarging the original threat model of SGX. Our sealing mechanism works without exposing the client's sealing keys outside their own enclave, without trusting third parties, including third-party enclaves, and works even when the sealing enclave is offline.

ACKNOWLEDGMENTS

This work was developed within the scope of the project nr. 51 "BLOCKCHAIN.PT - Agenda Descentralizar Portugal com Blockchain", financed by European Funds, namely "Recovery and Resilience Plan - Component 5: Agendas Mobilizadoras para a Inovação Empresarial", included in the NextGenerationEU funding program. The work was also supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UIDB/50021/2020 (INESC-ID) and 2021.07440.BD and by Universidade de Lisboa (ULisboa), Instituto Superior Técnico (IST), and Instituto de Engenharia de Sistemas e Computadores – Investigação e Desenvolvimento (INESC-ID).

REFERENCES

[1] Eurostat, "Cloud computing - statistics on the use by enterprises," https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises, Dec. 2021, online, Accessed: 2022-12-13.

[2] Gartner, "Gartner forecasts worldwide public cloud end-user spending to reach nearly $600 billion in 2023," https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023, Oct. 2022, press Release, Online, Accessed: 2022-12-13.

[3] F. Rocha and M. Correia, "Lucy in the sky without diamonds: Stealing confidential data in the cloud," in *1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments*, 2011, pp. 129–134.

[4] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (HotCloud)*, Jun. 2009.

[5] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-sealed data: A new abstraction for building trusted cloud services," in *Proceedings of the 21st USENIX Security Symposium (SEC)*, Aug. 2012.

[6] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Jun. 2013.

[7] D. Kaplan, J. Powell, and T. Woller, *AMD Memory Encryption*, AMD, Apr. 2016, release version 2021-10-18.

[8] *ARM Security Technology – Building a Secure System using TrustZone Technology*, Arm Limited, Apr. 2009, issue C.

[9] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, Jul. 2015.

[10] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Proceedings of the 17th International Middleware Conference*, Nov. 2016.

[11] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, May 2018.

[12] D. Vinayagamurthy, A. Gribov, and S. Gorbunov, "StealthDB: a scalable encrypted database with full SQL query support," in *Proceedings on Privacy Enhancing Technologies*, Mar. 2019.

[13] J. Wang, J. Wang, C. Fan, F. Yan, Y. Cheng, Y. Zhang, W. Zhang, M. Yang, and H. Hu, "SvTPM: SGX-based virtual trusted platform modules for cloud computing," *IEEE Transactions on Cloud Computing*, vol. 00, no. 00, Feb. 2023.

[14] I. Anati, S. Gueron, S. Johnson, and V. 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 (HASP)*, Jun. 2013.

[15] *Intel Software Guard Extensions Developer Guide*, Intel Corporation, Nov. 2022, release v2.18 for Linux.

[16] *Attestation Service for Intel Software Guard Extensions: API Documentation*, Intel Corporation, 2020, revision 6.0.

[17] *Intel Software Guard Extensions Developer Reference*, Intel Corporation, Nov. 2022, release v2.18 for Linux.

[18] J. Mechalas, *Code Sample: Intel Software Guard Extensions Remote Attestation End-to-End Example*, Intel Corporation, Apr. 2018.

[19] D. A. McGrew, "An interface and algorithms for authenticated encryption," RFC Editor, RFC 5116, Jan. 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5116.txt

[20] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. McKeen, *Intel Software Guard Extensions: EPID Provisioning and Attestation Services*, Intel Corporation, 2016.

[21] The Tamarin Team, *Tamarin-Prover Manual*, May 2022.

[22] *Intel Software Guard Extensions (Intel SGX) SDK for Linux OS: Developer Reference*, Intel Corporation, Nov. 2019, release v2.7.1 for Linux.

[23] "`dlopen`(3)," https://manpages.ubuntu.com/manpages/bionic/en/man3/dlopen.3.html, manpage of Ubuntu 18.04.

[24] "`dlsym`(3)," https://manpages.ubuntu.com/manpages/bionic/en/man3/dlsym.3.html, manpage of Ubuntu 18.04.

[25] "`clock_gettime`(2)," https://manpages.ubuntu.com/manpages/bionic/en/man2/clock_gettime.2.html, manpage of Ubuntu 18.04.

[26] *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives*, Intel Corporation, 2018.

[27] G. Chen, Y. Zhang, and T.-H. Lai, "OPERA: Open remote attestation for intel's secure enclaves," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2019.

[28] G. Chen and Y. Zhang, "MAGE: Mutual attestation for a group of enclaves without trusted third parties," in *Proceedings of the 31st USENIX Security Symposium*, Aug. 2022.

[29] Apache Software Foundation, "Apache Teaclave (incubating)," https://teaclave.apache.org/, online, Accessed: 2022-12-22.

[30] J. Park, S. Park, B. B. Kang, and K. Kim, "eMotion: An SGX extension for migrating enclaves," *Computers & Security*, vol. 80, Sep. 2019.

[31] 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 24th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2016.

[32] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li, "Secure live migration of SGX enclaves on untrusted cloud," in *Proceedings of the 47th IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, Jun. 2017.

[33] H. Liang, Q. Zhang, M. Li, and J. Li, "Toward migration of SGX-enabled containers," in *2019 IEEE Symposium on Computers and Communications (ISCC)*, Jun. 2019.

[34] K. Nakashima and K. Kourai, "MigSGX: a migration mechanism for containers including SGX applications," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, Dec. 2021.

[35] J. Guerreiro, R. Moura, and J. N. Silva, "TEEnder: SGX enclave migration using HSMs," *Computers & Security*, vol. 96, Sep. 2020.

[36] F. Alder, A. Kurnikov, A. Paverd, and N. Asokan, "Migrating SGX enclaves with persistent state," in *Proceedings of the 48th IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, Jun. 2018.