MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties

Guoxing Chen
Shanghai Jiao Tong University
guoxingchen@sjtu.edu.cn

Yinqian Zhang[*] ✉
Southern University of Science and Technology
yinqianz@acm.org

Abstract

Remote attestation mechanism enables an enclave to attest its identity (which is usually represented by the enclave's initial code and data) to another enclave. To verify that the attested identity is trusted, one enclave usually includes the identity of the enclave it trusts into its initial data in advance assuming no trusted third parties are available during runtime to provide this piece of information. However, when mutual trust between these two enclaves is required, it is infeasible to simultaneously include into their own initial data the other's identities respectively as any change to the initial data will change their identities, making the previously included identities invalid. In this paper, we propose MAGE, a framework enabling a group of enclaves to mutually attest each other without trusted third parties. Particularly, we introduce a technique to instrument these enclaves so that each of them could derive the others' identities using information solely from its own initial data. We also provide an open-sourced prototype implementation based on Intel SGX SDK, to facilitate enclave developers to adopt this technique.

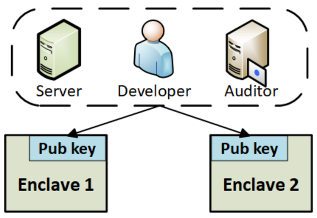
1 Introduction

As storage and computation outsourcing to clouds become more and more prevalent, cautious users and security researchers raise questions on whether the cloud providers could keep their data private and execute their applications as expected. Trusted execution environments (TEEs) have the potentials to offer efficient solutions to these concerns. A TEE is a secure area (usually called *enclave*) of a processor that protects the confidentiality and integrity of the code and data operated inside. Examples of TEEs include Intel Software Guard Extensions (SGX), AMD Secure Encrypted Virtualization (SEV), ARM TrustZone, Keystone [19], and Penglai [10].

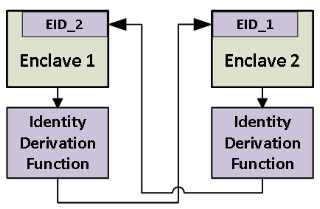
Trusting an enclave via remote attestation. Before interacting with an enclave, *e.g.*, outsourcing sensitive data for processing, the enclave must be "trusted". This trust is established via remote attestation [15]. In the context of attestation, the enclave is denoted the *attester* and the entity that wishes to establish trust on the attester is denoted the *verifier*, which could be either the user of the enclave (*i.e.*, a human empowered by code) or *another enclave*. Any software component between them can be considered untrusted or even malicious. The establishment of trust can be achieved by answering the following three questions:

- *Is the attester an enclave?* The root of trust of the remote attestation procedure lies in some specific component within the processor, *e.g.*, the processor's e-fuse with a *root secret* burnt into it. This root of trust identifies the underlying hardware and can be used to endorse a particular private key, called *attestation key*, which is used to sign the *evidence* generated by an attester to prove to a verifier that the attester is indeed an enclave running on an authentic TEE platform. Hence, when the evidence is verified to be valid, the verifier can be assured that the attester is indeed an enclave.
- *What is its identity?* The next immediate task for the verifier is to figure out the identity of the attester enclave. Usually, the attester enclave's identity is inserted by the TEE platform to the signed evidence sent to the verifier so that the verifier obtains the attester enclave's identity simultaneously when verifying the evidence. One enclave can be identified by its *initial code and data*. Besides including the whole initial code and data in the evidence, one alternative is to use the *enclave measurement*, which is the cryptographic hash of the initial code and data of an enclave.
- *Is the identity trusted?* After the verifier is convinced that it is communicating with an authentic enclave with its identity, the verifier needs to determine whether an enclave with the given identity can be trusted since a malicious application running inside an enclave could still leak sensitive data. Usually, the verifier compares the enclave identity `EID_rcv` from the received evidence with the ex-

[*]Corresponding author.



(a) Mutual attestation w/ TTP.



(b) Mutual attestation w/o TTP.

Figure 1: Mutual attestation w/ and w/o TTP.

pected one `EID_exp` from some reliable source. If they are equal, the verifier can be assured that the attester enclave is trusted. The reliable source can be the verifier's own memory or a trusted third party (TTP) such as a trusted enclave developer or any other trusted entity that maintains lists of trusted enclaves' identities.

Trusting multi-enclave applications via mutual attestation. The aforementioned trust establishment covers the simplest single-enclave applications. When a TEE application becomes more complicated and consists of multiple enclaves, it is very common that two enclaves need to interact with each other, where *mutual attestation* may be necessary. Mutual attestation is a mechanism that allows the communicating enclaves to attest each other for establishing a trust relationship. This is necessary, for example, in the following scenarios:

- Secure software development life-cycle: an enclave application is modularized into multiple components, each of which is loaded into a separate modular enclave. These modular enclaves need to verify each other's identities before interaction.
- Secure connection within distributed applications: when a client-server application is equipped with TEEs to protect sensitive code and data, the enclave on the server-side and the enclave on the client-side need to mutually attest each other to establish a secure communication channel.
- Secure interaction among mutually distrusting parties: when data needs to be exchanged between TEE-enhanced blockchain applications, each enclave should be able to attest the other's identity before accepting the transaction executed by that enclave.

Mutual attestation without TTPs. Mutual attestation can be achieved by simply performing attestation twice, one per each direction, by a trusted user. The user may also delegate this effort to a TTP. As shown in Fig. 1a, a TTP could be a stand-alone server that performs remote attestation with each enclave, validates the results, and exchanges secrets with the two enclaves as a middle man to bootstrap the trust. A TTP could also be a trusted developer that issues certificates for these enclaves or a trusted auditor that audits and signs these enclaves [28]. Such TTP-based solutions hard-code the corresponding public keys into the enclaves, enabling them to verify other enclaves' certificates or signatures to establish the trust.

However, integrating a trusted user or a TTP into the application's operation dramatically increases the trusted computing base (TCB) of the entire application. The security of the application will hinge upon the trustworthiness of the software stack of the user or the TTP, rather than solely the security of the enclave code itself (and, of course, that of the CPU hardware). Therefore, it is often desired to perform mutual attestation without TTPs.

In this paper, we aim to provide a mechanism for a group of enclaves to mutually attest one another by their enclave identities. However, we found this problem non-trivial. As shown in Fig. 1b, consider the cases of mutual attestation with two enclaves that would establish mutual trust with each other. The difficulty to do so lies in addressing the third question described above (*Is the identity trusted?*) for both enclaves. Without a TTP, both enclaves need to wait for the other enclave's identities to be finalized before they could include them in their initial data to finalize their identities and release them. The situation has some similarities to the deadlock problem, in that both parties wait on the data held by each other before they can proceed. This problem was first pointed out by Beekman *et al.* [3]. And a strawman solution of combining these enclaves into one single large enclave is discussed. To our best knowledge, no prior work has addressed this problem given enclaves with different identities.

Our solution: MAGE. The key challenge of mutual attestation for a group of enclave without TTPs is to enable each of these enclaves to obtain the identities of other enclaves in the same group from its own enclave memory so that during the attestation phase, the enclave could verify whether the identity of the attester is the same as one of the trusted enclaves in the same group. As such, we propose a framework, dubbed MAGE, to allow a group of enclaves to derive the identities of other enclaves in the group. The idea of MAGE is to split each enclave into two parts: the specific part that reflects the enclave's functionalities and the common part that holds the information for deriving the identities of enclaves in the same group.

We then provide a detailed design for TEEs that use the measurement as the identity. Particularly, our design allows each enclave to derive the measurements of enclaves in the same group from some *intermediate states* instead of the final outputs of the enclaves' measuring processes. The key observation is that the measurement calculation is deterministic and sequential. Knowing intermediate states and information to perform subsequent measuring operations would be sufficient to derive the final output, *i.e.*, the enclave measurement. When designed carefully, the problem could be resolved as all enclaves could generate intermediate states of their measuring processes and share them with others simultaneously.

We have implemented a prototype of MAGE using Intel SGX. Particularly, MAGE reserves at the end of each enclave a specific data segment with the same content which includes the intermediate hash value of each trusted enclave's content right before the reserved data segment. Hence, during runtime, each enclave knows the intermediate hash value of another trusted enclave (retrieved from the reserved data segment) and the content left to be added (*i.e.*, the reserved data segment), and thus could derive the other trusted enclave's measurement. The evaluation suggests that to enable mutual attestation for up to 85 enclaves, 62 KB enclave memory overhead for each enclave is introduced and roughly 21.7μs is needed to derive one measurement. While only a prototype on Intel SGX is presented in this paper, the method can be easily extended to different types of TEEs, *e.g.*, AMD SEV, and even between different types of TEEs, as long as they adopt similar mechanisms for the calculation of measurements.

We have already open-sourced MAGE on Github (https://github.com/donnod/linux-sgx-mage).

Paper outline. Sec. 2 presents motivating scenarios and Sec. 3 gives an overview of the proposed scheme. The main component, the technique for enclaves to mutually derive each other's identities is presented in Sec. 4. We present a prototype implementation and evaluate the performance in Sec. 5. Sec. 6 describes a case study and Sec. 7 discusses improvements and extensions. Sec. 8 presents related works and Sec. 9 concludes this paper.

2 Motivating Scenarios

In this section, we present three scenarios where mutual attestation is needed, describe potential TTP-based solutions, and discuss the benefits of alternatively adopting MAGE, a solution without TTPs.

2.1 Secure Software Development Life-cycle

Complex enclave applications have large TCB. A good practice of secure software development is to modularize the enclave application into multiple components and adopt privilege separation among these components to reduce the loss once some components are compromised. For example, consider a TEE-based smart home application that receives voice commands to operate smart devices, *e.g.*, asking the application to open the light, set the room temperature, and/or play the desired music. After obtaining voice inputs using voice sensors, the general process within the enclave application includes two steps: one is to extract commands from the voice inputs and the other is to execute the extracted commands. The former step could access sensitive biometrics information about the host, while the latter step focuses on executing the commands. When packing these two steps into one single enclave whose TCB will grow as more functionalities are added, vulnerabilities found in the second step might be leveraged to leak the sensitive biometrics from the first step. Enforcing software modularization by splitting these two steps into two enclaves could help reduce the severity of vulnerabilities. On the other hand, these modular enclaves require a mechanism to establish trust in each other. As in the smart home example, the enclave that processes voice inputs needs to verify the identity of the other enclave that executes the commands before releasing the host's commands while the latter needs to verify the identity of the former to ensure the commands to execute are indeed from the host.

TTP solutions with *Trusted Users*. When the user could act as the TTP, she could provide the identities of these modular enclaves to each other. After these modular enclaves are authenticated by the trusted user, they could obtain a list of trusted identities from the trusted user and establish secure channels with enclaves whose identities are in the list. As in the smart home example, the user could register with the input-processing enclave and the command-executing enclave, and act as a bridge to establish a secure channel between them.

MAGE helps reduce applications' TCB. TTPs are common targets for attackers. TTP solutions with trusted users require the related code on the user side to be secure and the user to operate correctly, which might become weaknesses of the system compared with the application code running within enclaves, especially when the application becomes more complicated and the TCB of the TTP increases. Hence, removing TTPs from the design, *e.g.*, using MAGE, could help keep the TCB smaller.

2.2 Secure Connection within Distributed Applications

Distributed applications usually run on multiple computers and communicate over a network. A typical distributed application structure is the client-server model. When both the server and the client are equipped with secure enclaves to protect sensitive code and data, a secure channel between the enclave on the server-side and the enclave on the client-side is desired to enable two-way authentication and secret provisioning. One example is OPERA, an Open Platform for Enclave Remote Attestation that provides distributed and privacy-preserving remote attestation services to Intel SGX enclaves [6]. It has two types of enclaves: issuing enclaves working as *servers* that are responsible for provisioning attestation keys to the other type of enclaves called attestation enclaves, which function as *clients*. The attestation enclave then uses the provisioned attestation key to provide attestation services to local enclaves. Both enclaves need to attest each other before providing attestation services. Particularly, the issuing enclave needs to authenticate the identity of the attestation enclave to prevent that any attestation key is leaked to any untrusted party while the attestation enclave should verify the identity of the issuing enclave to ensure that it only uses trusted attestation keys to provide attestation services to local enclaves.

TTP solutions with *Trusted Developers*. When the TEE application's developer is trusted, the developer's public key can be used to establish such mutual trust. Particularly, these enclaves could verify that they share the same developer's public key and create secure channels for communication. As in the OPERA example, if the developer can be trusted, her public key can be used by the issuing enclave and the attestation enclave to establish mutual trust.

MAGE enables a new level of trust on already-published software. For TTP solutions with trusted developers, since the developer's public key might be shared by all enclaves developed by the same developer, the trustworthiness of a TEE application could be affected by any future updates of other enclaves and even new enclaves from the same developer without any notification to the users. On the other hand, for already-published software which has been publicly verified, MAGE could help preserve its trustworthiness against any unverified future modification to the software.

2.3 Secure Interaction among Mutually Distrusting Parties.

Blockchains, *e.g.*, Bitcoin and Ethereum, enable mutually distrusting parties to interact, e.g., fulfill payments, without relying on any centralized TTPs. TEE has been leveraged to enhance existing blockchain applications. Designs that use SGX to provide privacy-preserving smart contracts have been proposed, such as Ekiden [7] and FastKitten [8], with different focuses. Ekiden provides efficient off-chain execution of single-round contracts, while FastKitten targets efficient off-chain execution of reactive multi-round contracts. Interactions between these different smart contracts could bring forth new applications. Consider a TEE-based online poker game implemented on FastKitten which requires a deposit for each player to join the poker game, and a credit scoring system implemented on Ekiden that manages players' credit scores. The interactions between these two applications could be as follows: players with higher scores in the credit scoring system are allowed to join an online poker game with smaller deposits and honest players in the games could gradually improve their credit scores in the credit scoring system. To enable such interactions between the enclaves of these smart contracts, mutual trust needs to be established, *i.e.*, each enclave should be able to attest the other's identity before accepting the transaction executed by that enclave.

TTP solutions with *Public Key Infrastructures*. When there exists a public key infrastructure (PKI) that is trusted by all participants, these enclaves could communicate with the PKI to obtain lists of trusted identities. As in the online poker game example, both enclaves could register with the PKI and fetch the other's identity from the PKI to establish mutual trust.

MAGE eliminates the need to run PKIs. Establishing and maintaining a PKI is challenging [9]. Especially for blockchain applications running across different countries, running a global secure PKI requires various skills and considerable resources. Alternatively, MAGE eliminates the need to run PKIs, facilitating more applications for mutually distrusting parties to interact.

3 Overview

In this section, we describe the threat model we assume, the problem we aim to address, and the overall workflow of MAGE.

3.1 Threat Model

We assume the hardware implementation of TEE and the remote attestation service provided by the TEE manufacturer are secure. That is, an adversary cannot breach the confidentiality and integrity of the enclave code and data, nor collude with the TEE-manufacturer-backed remote attestation service to endorse a malicious enclave.

We assume the code running inside of the enclaves is logically sound and secure against memory corruption attacks [4, 20, 29] and access-pattern-driven micro-architectural side channel attacks [5,11,14,21,24,26,30,31,34]. We assume the developers are honest during the software development and consider detecting attacks from the developer side, *e.g.*, including backdoors, is beyond the scope of this paper. We also leave assumptions about the user to the developers since different applications might have diverse threat models. Hence, to securely use TEE, software programs must be thoroughly examined to be free of such vulnerabilities [32, 33].

However, we assume TEE platforms are not trusted and may be controlled by the adversary. Specifically, the adversary controls all software components outside the enclave, including the operating system, the virtual machine manager (if any), *etc*. The adversary is also able to launch any enclave as she wants; however, she cannot create a malicious enclave whose identity is the same as a trusted enclave. Moreover, the adversary can perform man-in-the-middle attacks against the communication protocols between the enclaves, including but not limited to intercepting, dropping, replaying communications between any two enclaves.

3.2 Problem Formulation

In this paper, we aim to address the problem of enabling a group of enclaves to mutually attest one another without TTPs. These enclaves could be developed by the same developer or multiple different developers, and the interaction between them is also specified by the code. These enclaves could be run on the same TEE platform (*i.e.*, machine) or different

TEE platforms; they need to mutually attest each other before they could start to interact and/or collaborate.

Existing remote attestation mechanisms enable one enclave to verify that it is communicating with another actual enclave whose identity is in the received evidence. Hence, the key challenge of mutual attestation is for each enclave to obtain the expected identities of the other trusted enclaves without TTPs.

Consider a group of N enclaves $\texttt{Encl}_\texttt{i}$ $(i = 1,\ldots,N)$, each of which has an identity $ID(I_i)$, generated using a function $ID()$ from its initial content I_i which includes its initial code and data. When any two of these enclaves, denoted as $\texttt{Encl}_\texttt{i}$ and $\texttt{Encl}_\texttt{j}$, would like to establish mutual trust, both of them need to obtain the other's expected identity from some reliable source. Without a TTP to provide $\texttt{Encl}_\texttt{j}$'s identity to $\texttt{Encl}_\texttt{i}$, $\texttt{Encl}_\texttt{i}$ has to derive $\texttt{Encl}_\texttt{j}$'s identity by itself, *e.g.*, by hard-coding $\texttt{Encl}_\texttt{j}$'s identity in its initial data. Vice versa, $\texttt{Encl}_\texttt{j}$ also needs to be able to derive $\texttt{Encl}_\texttt{i}$'s identity.

Simply hard-coding the other enclave's identity in the enclave memory is not feasible. If we first hard-code $\texttt{Encl}_\texttt{i}$'s identity into $\texttt{Encl}_\texttt{j}$'s initial data. Then we finalize the identity of $\texttt{Encl}_\texttt{j}$ and try to hard-code it into $\texttt{Encl}_\texttt{i}$'s initial data. However, this will change $\texttt{Encl}_\texttt{i}$'s identity. The previously hard-coded $\texttt{Encl}_\texttt{i}$'s identity in $\texttt{Encl}_\texttt{j}$'s initial data will become incorrect.

The key insight in resolving the cyclic dependency is that each enclave $\texttt{Encl}_\texttt{i}$ can be split into two parts: (1) the specific part I'_i reflecting the functionalities of $\texttt{Encl}_\texttt{i}$ and (2) the common part I_{common} representing the content related to deriving the identities of all these N enclaves including $\texttt{Encl}_\texttt{i}$ itself. With such a split, the developers could focus on developing the enclaves' functionalities and leave the mutual attestation related components to our proposed solution.

Now we formally defines the mechanism for *mutual identity derivation*:

Definition 1 *Consider a group of N enclaves $\texttt{Encl}_i$ $(i = 1,\ldots,N)$, each of which has a specific part I'_i, a mechanism for mutual identity derivation consists of three functions ($\mathcal{G}$, $\mathcal{C}$, $\mathcal{F}$):*

- *$\mathcal{G}$ is called common part generation function that is used to generate the common part needed for deriving identities of these enclaves. It takes as input the specific parts of all N enclaves,* i.e., *$I'_1, \ldots, I'_N$, and outputs the common part for these enclaves, denoted as $I_{common} = \mathcal{G}(I'_1,\ldots,I'_N)$.*
- *$\mathcal{C}$ is called content builder function that is used to build the final content of $\texttt{Encl}_i$ from the specific part and the common part. It takes as input the specific part I'_i and the common part I_{common}, and outputs the final content of $\texttt{Encl}_i$ as $I_i = \mathcal{C}(I'_i, I_{common})$.*
- *$\mathcal{F}$ is called identity derivation function that is used for deriving identities from the common part. It takes as input the common part I_{common} and an index i $(= 1,\ldots,N)$, and outputs the identity of $\texttt{Encl}_i$. Specifically, $\mathcal{F}$ satisfies*

$$\mathcal{F}(I_{common}, i) = ID(I_i), \forall i = 1,\ldots,N \tag{1}$$

3.3 Workflow of MAGE

We now describe the workflow of MAGE, a framework enabling mutual attestation for a group of enclaves without TTPs, given a mechanism for mutual identity derivation ($\mathcal{G}$, $\mathcal{C}$, $\mathcal{F}$).

1. Develop a system including a group of enclaves that need to interact or collaborate, particularly the specific parts of these enclaves;
2. During compilation, derive the common part from the specific parts of these enclaves and build the final content of each enclave;
3. During runtime, enclaves derive the identities from the common part to be used in establishing mutual trust.

For the first step, a group of trusted enclaves that need to establish mutual trust are developed, especially when sensitive data needs to be transferred from one enclave to another. These enclaves could be developed by one or multiple developers. The algorithms about how the transferred secrets will be processed are up to the enclave developers, thus out of scope of this paper. The protocols for establishing secure channels include remote attestation [12, 25]. MAGE will provide application programming interfaces (APIs) that implement the identity derivation function $\mathcal{F}$ to be included in the attestation flows within the enclaves. The missing components are the common part of the enclaves in the group to be used by the identity derivation function to derive identities of other enclaves.

Then, during compilation, a tool provided by MAGE, that implements the corresponding common part generation function $\mathcal{G}$ and the content builder function $\mathcal{C}$ will be leveraged to extract the common part from the specific parts of these enclaves and build the final contents of these enclaves from the generated common part and the specific parts. Then, each enclave is ready to be released. Since this step requires all specific parts for generating the common part, MAGE does not scale well for a system with enclaves developed by different parties.

Lastly, during runtime, whenever the identity of a particular enclave in the group is needed, the identity derivation API will be called to derive the identity from the common part.

As most part of the workflow could be fulfilled using existing SDKs and protocols, the missing and most critical component is the mechanism for mutual identity derivation, *i.e.*, the common part generation function $\mathcal{G}$, the content builder function $\mathcal{C}$ and the identity derivation function $\mathcal{F}$. Since these three functions are also included in the TCB of the resulting system, their implementations should be thoroughly examined.

4 MAGE Design for Measurement-based Identities

In this section, we first describe the measurement-based identity which is widely used in existing TEE designs, *e.g.*, Intel SGX, AMD SEV, Keystone and Penglai, *etc*. We then introduce a design of MAGE that fits such TEEs. Finally, we take Intel SGX as an example to provide an instantiation of MAGE.

4.1 Measurement-based Identity

In existing TEE designs, the enclave measurement is usually used to identify enclaves. Generally, the enclave measurement is the cryptographic hash of the contents of an enclave, including initial code and data. Hence, the enclave user could verify the identity of an enclave by comparing only its measurement with an expected value.

A cryptographic hash function $\mathcal{H}$ is a mathematical algorithm that maps or transfers data of arbitrary size, usually called a *message* M, into a bit array of fixed size, called a *message digest* or a *hash value* $h = \mathcal{H}(M)$. It is a one-way function such that with a hash value, it is practically infeasible to compute the original message. And it is deterministic such that the same message always results in the same hash value.

To compute a hash value, an arbitrary-length message M is firstly broken into fixed-length blocks $M^1, \ldots, M^L$ (the last block should be length padded) and processed one by one sequentially. A fixed-length *internal state* (or *intermediate hash*) is updated after each block is processed. Particularly, an update function $\mathcal{H}_{\text{upd}} : (state, data) \longrightarrow state$ takes as input an internal state and a data block and outputs an updated internal state. After updating the last block, a finalization function $\mathcal{H}_{\text{fin}} : (state, data) \longrightarrow hash$ takes as input the latest internal state and a data block containing the length of the message and outputs the resulting hash value. Loosely speaking, the computation can be represented as follows:

$$\begin{aligned}\mathcal{H}(M) &= \mathcal{H}(M^1||\ldots||M^L)\\ &= \mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\ldots\mathcal{H}_{\text{upd}}(\text{IV}, M^1), \ldots), M^L), |M|)\end{aligned}$$

where IV represents the initial value of the internal state.

For simplicity, we abuse the notation of $\mathcal{H}_{\text{upd}}$ to consume multiple blocks by recursively applying the original $\mathcal{H}_{\text{upd}}$ as follows:

$$\begin{aligned}&\mathcal{H}_{\text{upd}}(state, B^1||\ldots||B^K)\\ =&\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\ldots\mathcal{H}_{\text{upd}}(state, B^1), \ldots), B^K)\end{aligned}$$

With such a deterministic and sequential computation process, the hash value of the concatenation of two (length-padded) messages X and Y, denoted as $X||Y$, can be calculated as follows:

$$\mathcal{H}(X||Y) = \mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\text{IV}, X), Y), |X| + |Y|)$$

4.2 Mutual Identity Derivation Mechanism for Measurement-based Identity

Now we introduce a mutual identity derivation mechanism for measurement-based identity. Consider an enclave `Encl_i` with content I_i who uses its measurement $\mathcal{H}(I_i)$ as its identity. Recall that the hash value calculation is deterministic and sequential. That is, when I_i is split into two length-padded segments I_i^a, I_i^b, we have

$$\mathcal{H}(I_i) = \mathcal{H}(I_i^a||I_i^b) = \mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a), I_i^b), |I_i^a| + |I_i^b|)$$

Hence, hard-coding the fixed-length $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a)$ (which we call a *pre-measurement* of `Encl_i`), the length of I_i^a, and the arbitrary-length I_i^b into the initial data of another enclave, say `Encl_j`, is sufficient for `Encl_j` to compute the measurement of `Encl_i`. Particularly, when I_i^b is empty, it resembles the case of hard-coding the measurement of `Encl_i` directly while when I_i^a is empty, it becomes hard-coding the entire content of `Encl_i` into `Encl_j`.

Since $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a)$ is fixed-length and is computed before the measuring process reaches I_i^b, if the value of $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a)$ and the length of I_i^a are included in I_i^b, any enclave with I_i^b only is able to derive the measurement of `Encl_i` by first retrieving the value of $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^a)$ and the length of I_i^a from I_i^b and then continuing the computation of the hash value with I_i^b. This intuition leads to our design of mutual identity derivation mechanism with measurement-based identity. The idea is to derive the internal states of these enclaves' specific parts and combine them to form the common part. Specifically, we have

- A common part generation $\mathcal{G}$ that calculates the internal states of the specific parts of these N enclaves along with the sizes of the specific parts to form an array as the common part:

$$\begin{aligned}I_{\text{common}} =&\mathcal{G}(I_1', \ldots, I_N')\\ =&[(\mathcal{H}_{\text{upd}}(\text{IV}, I_1'), |I_1'|), \ldots, (\mathcal{H}_{\text{upd}}(\text{IV}, I_N'), |I_N'|)]\end{aligned}$$

  where the i-th entry of the common part is a tuple of $I_{\text{common}}^i = (\mathcal{H}_{\text{upd}}(\text{IV}, I_i'), |I_i'|)$ and $I_{\text{common}}^i[j], j = 1, 2$ denotes the first or the second element of the tuple.

- A content builder function $\mathcal{C}$ that produces the final content by concatenating the specific part and the common part:

$$I_i = \mathcal{C}(I_i', I_{\text{common}}) = I_i'||I_{\text{common}}$$

- An identity derivation function $\mathcal{F}$ that generates the identity of `Encl_i` by retrieving the i-th entry from the common part and completing the hash value calculation:

$$\begin{aligned}&\mathcal{F}(I_{\text{common}}, i)\\ =&\mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(I_{\text{common}}^i[1], I_{\text{common}}), I_{\text{common}}^i[2] + |I_{\text{common}}|)\\ =&\mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\text{IV}, I_i'), I_{\text{common}}), |I_i'| + |I_{\text{common}}|)\\ =&\mathcal{H}(I_i'||I_{\text{common}}) = \mathcal{H}(I_i) = ID(I_i), \forall i = 1, \ldots, N\end{aligned}$$

4.3 MAGE for Intel SGX

We now use Intel SGX as an example to describe how to instantiate MAGE with actual TEEs. Intel Software Guard Extensions (SGX) is a new hardware feature introduced on recent Intel processors. To implement secure enclaves, Intel SGX reserves a specified range of DRAM, called *Processor Reserved Memory* (PRM), which will deny accesses from any software (including the operating system) other than the enclave itself. The enclave's code, data, and related data structures are stored in a subset of PRM, called *Enclave Page Cache* (EPC), which is further split into 4 KB EPC pages.

4.3.1 SGX Enclave Measurements

In the current Intel SGX design, enclave measurements are calculated using SHA-256 [2]. SHA-256 is a Secure Hash Algorithm (SHA) that is used for generating 256-bit digests of messages. The generated digests are used to protect the integrity of the messages. SHA-256 has three algorithms:

- *Initialization algorithm* initializes 8 32-bit words, as the initial value of the intermediate hash, before calculating the digest.
- *Update algorithm* takes a 512-bit block as input at a time and updates the intermediate hash using pre-defined compression functions.
- *Finalization algorithm* updates the intermediate hash with the last 512-bit block which contains the number of all bits that have been updated to the intermediate hash and produces the final 256-bit digest by concatenating the resulting 8 32-bit words.

The calculation of enclave measurement is performed along with the creation of the enclave, as shown in Fig. 2. Specifically,

1. When creating an enclave, the SGX instruction ECREATE will be called to create the first EPC page, called *SGX Enclave Control Structure* (SECS) page, which maintains the metadata of the enclave to be created, such as the base address, the size of enclave memory required, and the 256-bit enclave's measurement, *i.e.*, `MRENCLAVE`. The instruction ECREATE initializes the `MRENCLAVE` field using SHA-256 Initialization algorithm and updates its value using SHA-256 Update algorithm, which takes as input a 512-bit block including the enclave's metadata.
2. Then, via the SGX instruction EADD, two types of EPC pages will be added: (1) *Thread Control Structure* (TCS) pages that store information needed for logical processors to execute the enclave code; and (2) *Regular* (REG) pages that store the enclave code and data. When adding an EPC page, a 64-byte data structure, called *Security Information* (SECINFO), is also needed for the EADD instruction to specify the properties of the added EPC page, such as page type (a TCS page or a REG page), and access permissions (whether the page can be read, written and/or executed).

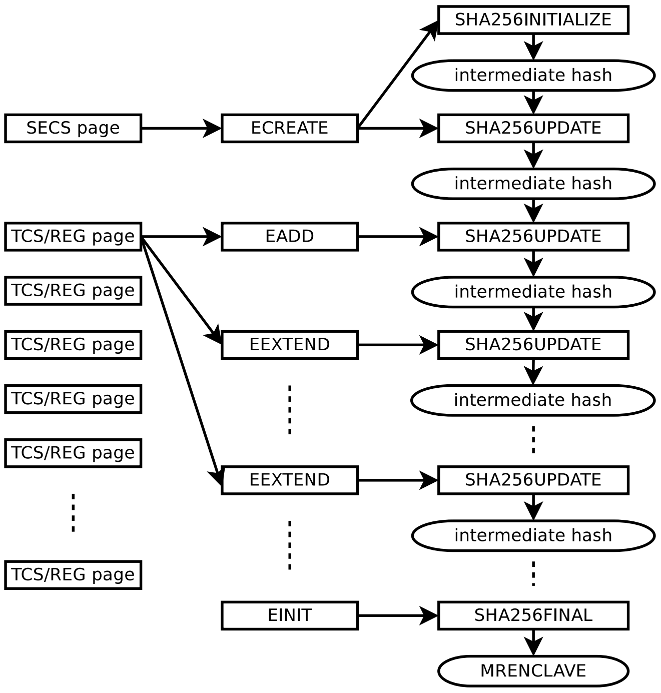


Figure 2: Enclave measurement calculation flow.

When EADD is called each time to create a TCS or REG page, it firstly updates `MRENCLAVE` with a 512-bit block including only the metadata of the page to be added, *e.g.*, its offset and access permissions. The content of the page is measured by the SGX instruction EEXTEND which measures 256 bytes at one time. For each 256 bytes of an EPC page, EEXTEND performs the SHA-256 Update algorithm 5 times. The first iteration measures a 512-bit block containing the metadata of the 256 bytes of data including its offset, and each of the following 4 iterations measures 64 bytes of the content. To measure a 4 KB EPC page, 16 EEXTEND operations are needed.

3. After all enclave pages are loaded, the SGX instruction EINIT will be invoked to finalize the creation of the enclave. EINIT finalizes the measurement using SHA-256 Finalization algorithm which updates `MRENCLAVE` the last time with a 512-bit block containing the total count of bits that have been updated into `MRENCLAVE`. This count is initialized by ECREATE and updated through ECREATE, EADD and EEXTEND. The enclave code could be run then.

4.3.2 Mutual Identity Derivation for Intel SGX

Now we describe a mutual identity derivation mechanism for Intel SGX. We will start with the case of two enclaves and then generalize to cases of multiple enclaves.

Generating and instrumenting the common part during development. The common part that is required for deriving the other enclave's measurement needs to be extracted (via

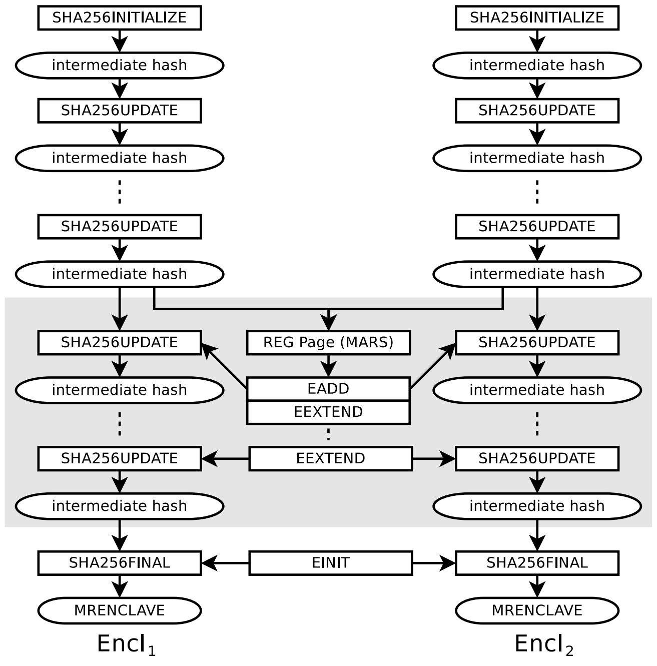


Figure 3: The flow of generating the common part and building the final content for mutual identity derivation.

Table 1: `MAINFO`: information needed for mutual identity (measurement) derivation.

Component	**Description**
`PREMR`	The intermediate hash before `MARS`;
`COUNT`	The number of bytes updated to `PREMR`;
`OFFSET`	The offset of `MARS`;
`SECINFO`	The security information of `MARS`;

the common part generation function) and hardcoded into the enclave's initial data (via the content builder function). It should be done during the enclave development phase. The whole flow is shown in Fig. 3. Basically, the development of $\texttt{Encl}_1$ and $\texttt{Encl}_2$ could follow the following steps:

- Each of these two enclaves reserves a data segment (called *mutual attestation reserved segment*, denoted as `MARS`) of the same size (*e.g.*, 4 KB, the size of one EPC page) for the common part I_{common} in their own enclave memory, which will be loaded last during the enclave creation. Further, this data segment should be aligned to the page boundaries, so that it will not overlap with other enclave pages during measurement calculation. The SHA-256 intermediate hash of all pages before this reserved region, *i.e.*, the pre-measurement (denoted as `PREMR`), will be calculated.
- For each enclave, the information needed for deriving its own measurement, called *Mutual Attestation Information* (`MAINFO`) as depicted in Table 1, is collected. Particular, `MAINFO` contains four fields: (1) the pre-measurement $\texttt{PREMR}_\texttt{i}$ of $\texttt{Encl}_\texttt{i}$, (2) the number of bytes updated to $\texttt{PREMR}_\texttt{i}$, (3) the offset of the reserved data segment $\texttt{MARS}_\texttt{i}$, and (4) the security information of $\texttt{MARS}_\texttt{i}$. The former two are used to reconstruct the state of measurement calculation before updating the reserved data segment $\texttt{MARS}_\texttt{i}$, and the latter two are needed for updating the $\texttt{MARS}_\texttt{i}$ into the hash value as described in Sec. 4.3.1. Particularly, `SECINFO` field can be dropped if a constant SECINFO is adopted, *e.g.*, by assuming `MARS` contains only read-only data pages. While fixing the offset of `MARS` could also save the memory space for the `OFFSET` field, it will add extra workload for enclave developers to adjust the enclave memory layouts, which might not be preferred.
- The collected `MAINFO`s of both enclaves are organized to form the common part to be instrumented into the `MARS` of these enclaves so that each enclave knows the `MAINFO` of the other enclave (from its own `MARS`) and the content of the other enclave's `MARS` (same as its own `MARS`).

Deriving measurements during runtime. As described in Sec. 4.3.1, the calculation of the enclave measurement depends on the order the enclave pages are created. Even with exactly the same enclave code and data, when loaded in different orders, different measurements will be generated. Hence, during enclave creation, EPC pages of $\texttt{Encl}_1$ and $\texttt{Encl}_2$ have to be in a particular order that can be simulated during runtime to derive their measurements, as shown in Fig. 3. Particularly, all enclave pages except `MARS`s need to be created in the same order that generates the pre-measurement `PREMR`. The `MARS` is created and loaded last. The enclave is initialized afterward. We will describe how to adjust the order the enclave pages are created in Sec. 5 when needed, and also discuss an alternative design when the loading order cannot be altered in Sec. 7. Now we assume the enclave is loaded exactly in the same order as how the `PREMR` is computed.

After one enclave, *e.g.*, $\texttt{Encl}_1$, is created, it could derive the measurement of the other enclave ($\texttt{Encl}_2$) according to the identity derivation function as follows:

- From $\texttt{Encl}_1$'s reserved data segment $\texttt{MARS}_1$, $\texttt{Encl}_1$ retrieves $\texttt{Encl}_2$'s pre-measurement $\texttt{PREMR}_2$, the number of bytes updated $\texttt{PREMR}_2$, the offset of $\texttt{Encl}_2$'s reserved data segment $\texttt{MARS}_2$, and the SECINFO of $\texttt{MARS}_2$.
- $\texttt{Encl}_1$ simulates $\texttt{Encl}_2$'s process of loading the reserved data segment $\texttt{MARS}_2$, whose content is the same as $\texttt{MARS}_1$ which $\texttt{Encl}_1$ could access directly. $\texttt{Encl}_1$ then updates the number of bytes contributing to the resulting SHA-256 intermediate hash and performs the finalization operation to obtain the measurement of $\texttt{Encl}_2$.

For verification, recall how $\texttt{Encl}_2$'s measurement is actually generated by the SGX implementation: The SHA-256 intermediate hash is updated as $\texttt{Encl}_2$'s pages are created one by one; When it comes to $\texttt{MARS}_2$ which will be loaded last, the SHA-256 intermediate hash is $\texttt{PREMR}_2$, assuming the correct loading order; The SHA-256 intermediate hash keeps being updated when loading $\texttt{MARS}_2$ and gets finalized afterward.

Algorithm 1: Identity Derivation Function

```
   Input: idx
   Output: mrenclave
1  if idx ≥ total number of MAINFO entries in MARS then
2      return NULL;
3  [PREMR, COUNT, OFFSET, SECINFO] ← idx-th MAINFO in MARS;
4  sha_handle ← sgx_sha256_init();
5  replace related fields of sha_handle with PREMR and COUNT;
6  for page in MARS do
7      sgx_sha256_update(sha_handle, “EADD”||OFFSET||SECINFO);
8      for every 2048-bit data in page do
9          sgx_sha256_update(sha_handle, “EEXTEND”||OFFSET);
10         sgx_sha256_update(sha_handle,data[511:0]);
11         sgx_sha256_update(sha_handle,data[1023:512]);
12         sgx_sha256_update(sha_handle,data[1535:1024]);
13         sgx_sha256_update(sha_handle,data[2047:1536]);
14     OFFSET = OFFSET + 256;
15 mrenclave ← sgx_sha256_get_hash(sha_handle);
16 return mrenclave;
```

Hence, what $\texttt{Encl}_1$ derives is exactly the measurement of $\texttt{Encl}_2$. Similarly, $\texttt{Encl}_2$ could also derive $\texttt{Encl}_1$'s measurement.

Supporting multiple enclaves. Now we describe how to extend the method presented above to a group of (more than two) enclaves, $\texttt{Encl}_1$, $\texttt{Encl}_2$,..., $\texttt{Encl}_N$. MAINFOs of all enclave are extracted and organized to generate the common part, *i.e.*, MARS, to be instrumented into these enclaves. And the creation of each enclave needs to follow the pre-defined order for calculating the corresponding PREMR. After one enclave is created, it could derive the measurement of any enclave by fetching the corresponding MAINFO from its own MARS and simulating the measuring process with the content of its own MARS. The identity derivation function is shown in Algorithm 1. It takes as input the index *idx* of the enclave measurement to be derived, and outputs the derived enclave measurement. The function retrieves the *idx*-th MAINFO to create an SHA-256 handle, updates it with the content of MARS following the process described in Sec. 4.3.1. `sgx_sha256_init()`, `sgx_sha256_update()` and `sgx_sha256_get_hash()` are the implementations of the initialization, update and finalization algorithms inside the enclave.

5 Implementation and Evaluation

In this section, we describe our prototype implementation of MAGE and evaluate its runtime performance overhead and memory overhead.

5.1 Implementation

MAGE is implemented by extending the Intel SGX SDK (version 2.6.100.51363) [16]. Specifically, it consists of three components: (1) An SDK library that reserves a data segment for MARS and provides APIs to support derivation of measurements from MAINFOs located in MARS; (2) A modified enclave loader that loads MARSs last when creating enclaves; (3) A modified signing tool that extracts MAINFO from an enclave and fills the MARS of an enclave with a list of extracted MAINFOs.

MAGE library. `libsgx_mage` is implemented to facilitate enclave developers to use MAGE. When included in an enclave, it reserves a read-only data section, named `.sgx_mage`, to be used as MARS. The range of the `.sgx_mage` section is aligned to page boundaries, *i.e.*, 4KB. So its size is a multiple of the page size, *i.e.*, 4KB. Besides reserving the `.sgx_mage` section, `libsgx_mage` provides two APIs:

- `sgx_mage_size()` examines the `.sgx_mage` section and returns the total number of MAINFOs in it.
- `sgx_mage_gen_measurement()` takes as input an index of the enclave whose measurement is requested and outputs the resulting measurement. Particularly, it retrieves from the `.sgx_mage` section the corresponding MAINFO specified by the index and calculates the final measurement following Algorithm 1.

Modified enclave loader. The original enclave loader in Intel SGX SDK loads enclave code and data pages first and then the TCS pages. Hence, the `.sgx_mage` section, as a data segment, will not be loaded last by default. To address this, we modified the enclave loader to load the enclave pages in two stages:

- First, the modified enclave loader follows the original loading process except that when an `.sgx_mage` section is encountered, it skips the `.sgx_mage` section. Here the `libsgx_mage` APIs are located in code pages and loaded along with the original enclave code pages.
- Second, when all other pages, including enclave code and data pages and the TCS pages, are loaded, the modified enclave loader checks whether there is an `.sgx_mage` section, and loads pages in the `.sgx_mage` section if found.

Particularly, if no `.sgx_mage` section is present, the modified enclave loader will load the enclave in the same order as the unmodified enclave loader, producing the same measurement. When there exists an `.sgx_mage` section, the modified and unmodified enclave loaders will produce different measurements due to different loading orders, as the unmodified enclave loaders will load `.sgx_mage` section ear-

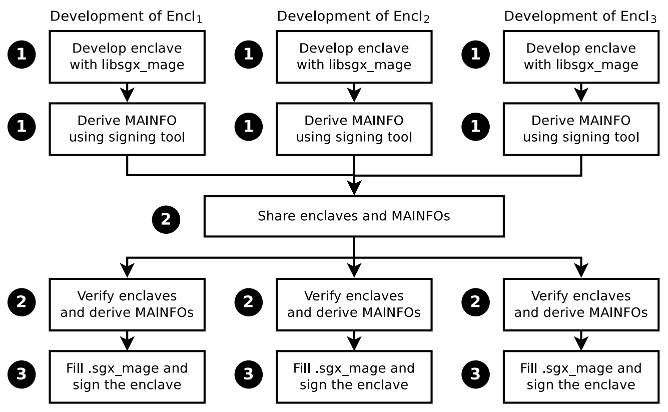

Development of Encl$_1$
Development of Encl$_2$
Development of Encl$_3$
Develop enclave with libsgx_mage
Derive MAINFO using signing tool
Share enclaves and MAINFOs
Verify enclaves and derive MAINFOs
Fill .sgx_mage and sign the enclave

Figure 4: Workflow of enclave development using MAGE.

lier than the modified one. Since our implementation of `sgx_mage_gen_measurement()` produces the measurement in the same order as the modified enclave loader, platforms running the enclaves developed with MAGE need to use the modified enclave loader. If using a modified loader is undesired, we discuss an alternative design in Sec. 7.

Modified signing tool. The original signing tool is provided by Intel SGX SDK for enclave developers to sign enclaves so that they can be accepted by the Intel-signed Launch Enclave and thus be launched successfully. The signing tool simulates the loading process of the enclave to calculate the measurement before signing it. We modified the signing tool to provide the following two functionalities:

- Deriving `MAINFO`s: given an enclave developed with MAGE, the modified signing tool could simulate the first stage of the modified enclave loader, which loads all pages except for the `.sgx_mage` section, to generate `MAINFO`, which includes the SHA-256 intermediate hash, *i.e.*, the `PREMR`, the number of bytes updated to `PREMR`, and the offset of the `.sgx_mage` section. The SECINFO is not included as our prototype implementation adopts a constant value of SECINFO with access permissions set to be read-only.
- Filling the `.sgx_mage` section: given an enclave developed with MAGE and a list of `MAINFO`s derived from a group of trusted enclaves, the modified signing tool could fill the `.sgx_mage` section with the list of `MAINFO`s. The measurement of the instrumented enclave will be recalculated and signed afterward.

As shown in Fig. 4, the workflow of enclave development using MAGE can be depicted as follows: ❶ each enclave is developed with the `libsgx_mage` library, and its `MAINFO` is derived using the modified signing tool. ❷ if these enclaves are from different developers, the enclave developers share their enclaves with one another, so that they could validate the trustworthiness of the enclaves from other developers, and then use the modified signing tool to derive the `MAINFO`s of them. ❸ with the same list of `MAINFO`s, the modified signing tool can be leveraged to fill the `.sgx_mage` section of each enclave, and sign the resulting enclave before release.

5.2 Evaluation

Now we describe the evaluation of our prototype implementation of MAGE. Results are measured on a Lenovo Thinkpad X1 Carbon (4-th Gen) laptop with an Intel Core i5-6200U processor and 8GB memory.

Since the results are highly related to the size of the `.sgx_mage` section, so we evaluate the metrics with regards to different sizes of the `.sgx_mage` section.

The number of `MAINFO`s supported. We first calculate the number of `MAINFO`s that can be stored in an `.sgx_mage` section with L bytes (L is a multiple of the page size, *i.e.*, 4KB). The content of an `.sgx_mage` section is organized as a structure as follows: the first 8 bytes hold the total number of `MAINFO`s and the rest is used to store the content of these `MAINFO`s. Each `MAINFO` takes 48 bytes (32-byte `PREMR`, 8-byte `COUNT`, and 8-byte `OFFSET`). `SECINFO` is not included as we use a constant SECINFO in our prototype implementation. Hence, $\lfloor \frac{L-8}{48} \rfloor$ `MAINFO`s can be supported. For example, an `.sgx_mage` section of one page size could support up to 85 `MAINFO`s. On the other hand, to support N `MAINFO`s, a total of $\lceil \frac{48N+8}{4096} \rceil$ pages are needed. For example, supporting $N = 10,000$ `MAINFO`s requires an `.sgx_mage` section of 118 pages (472 KB).

Efficiency of measurement derivation. We then measure the time needed for deriving one measurement. From the measurement derivation function described in Algorithm 1, we can see that the time needed for the derivation is independent of the size of the original content of the enclave and the actual number of `MAINFO`s in the `.sgx_mage` section. This is because the content is updated into a single `MAINFO` where the derivation process starts from and all bytes in the `.sgx_mage` section need to be updated into the final measurement.

So we evaluated the efficiency of measurement derivation using a dummy enclave with only one enclave function that calls `sgx_mage_gen_measurement()` to derive one measurement from its `.sgx_mage` section. Also, only one `MAINFO` from itself is generated and inserted into its `.sgx_mage` section. As expected, we verified that the derived measurement is the same as its own measurement. We measured the time (averaged from 10000 iterations) needed to run one invocation of `sgx_mage_gen_measurement()` when the number of pages in the `.sgx_mage` section ranges from 1 to 10000. When the `.sgx_mage` section has a size of a single page, the time for deriving one measurement is around 2.17e$-$5 seconds or 21.7μs. When the `.sgx_mage` section has a size of 10000 pages, the time for deriving one measurement is around 0.212 seconds. The time needed for deriving one measurement increases almost linearly with regards to the number of pages in the `.sgx_mage` section because the main operations are updating the intermediate hash value with the content of the `.sgx_mage` section.

Memory overhead. The memory overhead introduced by MAGE includes two components: (1) extra data pages for `MARS`; (2) extra code pages related to the measurement derivation. The first part is straightforward, which is the size of the `.sgx_mage` section. To calculate the second part, we created another enclave similar to the dummy enclave we just developed, except that the MAGE-related code and data are removed. We calculated the differences in memory sizes between these two enclaves. Subtracting the first component from the total extra memory, we got the size of the second component, which is around 58 KB. Since `libsgx_mage` leverages the SHA-256 implementations provided in the Intel SGX SDK, the second component could be smaller if the original enclave already includes them. Hence, the total memory overhead with a `.sgx_mage` section of n pages is $58+4n$ KB.

6 Case Study: OPERA with MAGE

We now give an example of integrating MAGE into an open-sourced SGX application, OPERA. We will first describe the design of OPERA and then introduce the integration of MAGE into OPERA.

6.1 OPERA: Open Platform for Enclave Remote Attestation

OPERA introduces an attestation service to provide better privacy guarantees to enclaves [6]. The proposed attestation service is based on the same scheme adopted by Intel, *i.e.*, Enhanced Privacy ID (EPID). EPID is a digital signature algorithm that could protect the anonymity of SGX platforms [17]. To facilitate EPID based remote attestation, Intel introduces two services, *i.e.*, Intel Provisioning Service (IPS) and Intel Attestation Service (IAS), and provides SGX platforms with two privileged enclaves, Intel-signed Provisioning Enclave (PvE) and Quoting Enclave (QE). Particularly, IPS and the PvE run an EPID provisioning protocol to provision an EPID private member key (attestation key) to an SGX platform. The EPID private member key could only be accessed by the PvE and the QE. Otherwise, any malicious enclave that could access the EPID private member key will be able to forge valid signatures to deceive the remote entity. Hence, to get a signature signed by the EPID private member key, the attester enclave needs to firstly attest itself to the QE via local attestation. After the QE verifies the attester's report, it will generate a data structure, called *quote*, which contains the attester's measurement and attestation data that are copied from the report, and sign the quote using the EPID private member key. The attestation enclave could then use the signed quote to attest itself to the remote entity. In the current design, the signed quote is encrypted by the QE, so that the remote entity has to forward the encrypted signed quote to IAS for verification, raising sensitive users' concerns about the enclave's privacy.

OPERA is proposed to address such privacy concerns. It has two types of enclaves: issuing enclaves (`IssueE`) that are responsible to provision EPID private keys to the other type of enclaves called attestation enclaves (`AttestE`). `AttestE`s then use the provisioned EPID private keys to provide attestation services to local enclaves. One important property of OPERA is its *openness*, *i.e.*, the implementation is completely open so that its code (and hence behaviors) can be publicly verified and thus is trustworthy, while its developer can be untrusted. This property enables OPERA to achieve better security without introducing extra trusted parties. As such, mutual attestation without TTPs is desired in OPERA.

However, due to the lack of a mutual attestation mechanism without TTPs between `IssueE` and `AttestE`, the authors of OPERA provided an alternative design that transfers part of the attestation workload to the user of the system. Particularly, only `IssueE` verifies the identity of `AttestE` before provisioning EPID private keys. `AttestE` has no means to verify whether the provisioned EPID private keys are from a trusted enclave or not. The TCB of the OPERA includes partial code on the user side that verifies whether `AttestE` obtained the EPID private keys from a trusted `IssueE` or not. While the authors proved the secrecy property of the protocol using ProVerif (an automatic cryptographic protocol verifier), they did not discuss other potential threats due to the lack of mutual attestation. For example, the adversary could provision the `AttestE` with an EPID private key controlled by the adversary and launch co-location attacks on the user of OPERA by monitoring the error message (*e.g.*, "EPID private key is from an untrusted server enclave") of the attestation results. Hence, designing a mutual attestation mechanism for OPERA could reduce the attack surface and make it more self-contained.

6.2 OPERA-MAGE

We integrated OPERA with MAGE, resulting in OPERA-MAGE. The integration includes two parts:

- Add identity verification logic to `AttestE`: In the existing OPERA design, only `IssueE` verifies the identity of `AttestE` before provisioning an EPID key while `AttestE` is not able to verify the identity of `IssueE`. We modified `AttestE` to include the verification logic so that both `IssueE` and `AttestE` could mutually attest each other before the EPID provisioning process.
- Include MAGE in both `IssueE` and `AttestE`: We included the MAGE library in both `IssueE` and `AttestE` and used the modified signing tool to extract the common part, finalized `IssueE` and `AttestE`, and signed them.

We run OPERA-MAGE on an SGX platform with the modified enclave loader installed and the resulting `IssueE` and `AttestE` could successfully attest each other and provide the claimed attestation service to other local enclaves. We have already open-sourced OPERA-MAGE on Github (https://github.com/donnod/opera-mage).

7 Discussion

In this section, we discuss possible extensions of MAGE, its limitations, and alternative designs.

7.1 Extending MAGE with Untrusted Storage

The basic design of MAGE enables the derivation of identities from information completely inside the enclave memory. Next, we discuss how MAGE can be extended with untrusted storage outside the enclaves (*e.g.*, unencrypted memory, hard drives, *etc*.).

Supporting unmodified enclave loaders. In our implementation with Intel SGX, we modified the enclave loader to rearrange the order of enclave pages to be loaded in a way such that the common part is loaded after all other content. When such enclave loader modification is undesired, we present a potential solution to support MAGE without the need of modifying enclave loaders.

Consider the specific part of an enclave I_i' is split into two parts (I_i^{pre}, I_i^{post}) where I_i^{pre} is the part loaded before I_{common}, and I_i^{post} is the part loaded after I_{common}. The measurement can be calculated as

$$\begin{aligned}&\mathcal{H}(I_i^{pre}||I_{\text{common}}||I_i^{post})\\ =&\mathcal{H}_{\text{fin}}(\mathcal{H}_{\text{upd}}(\mathcal{H}_{\text{upd}}(\text{IV}, I_i^{pre}), I_{\text{common}}||I_i^{post}),\\ &\qquad\qquad |I_i^{pre}| + |I_{\text{common}}| + |I_i^{post}|)\end{aligned}$$

We can still insert the intermediate hash of all pages before I_{common}, *i.e.*, $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^{pre})$ and the length of I_i^{pre} into I_{common}. As for I_i^{post}, storing all its content within I_{common} might introduce too much memory overhead. When untrusted storage is available to hold the content of I_i^{post}, the enclave could access it during runtime to derive the measurements. The integrity of I_i^{post} could be examined by storing its hash value within I_{common}. The common part generation function can be defined as

$$\begin{aligned}I_{\text{common}} =& \mathcal{G}((I_1^{pre}, I_1^{post}), \dots, (I_N^{pre}, I_N^{post}))\\ =& [(\mathcal{H}_{\text{upd}}(\text{IV}, I_1^{pre}), |I_1^{pre}|, \mathcal{H}(I_1^{post})),\\ &\vdots\\ &(\mathcal{H}_{\text{upd}}(\text{IV}, I_N^{pre}), |I_N^{pre}|, \mathcal{H}(I_N^{post}))]\end{aligned}$$

During runtime, the enclave requests the host program to provide the content of I_i^{post} for measurement derivation. The hash value $\mathcal{H}(I_i^{post})$ stored in I_{common} will be used to verify the integrity of I_i^{post} and obtain its length $|I_i^{post}|$. Then, the measurement derivation function could start from $\mathcal{H}_{\text{upd}}(\text{IV}, I_i^{pre})$ and update the content of I_{common} and I_i^{post} into the intermediate hash to obtain the final measurement. This design requires extra untrusted storage to store I_i^{post}, which unfortunately may take longer time to derive a measurement when I_i^{post} is large.

Increasing scalability. In our current design, the time cost for identity derivation and memory overhead grows linearly with regards to the size of the common part which reflects the group size of trusted enclaves. To support a larger number of enclaves, when untrusted storage outside the enclave is available, the content of the pre-measurement array can be moved out of the common part and only its hash value is stored within the common part for integrity protection:

$$I_{\text{common}} = \mathcal{H}([(\mathcal{H}_{\text{upd}}(\text{IV}, I_1'), |I_1'|), \dots, (\mathcal{H}_{\text{upd}}(\text{IV}, I_N'), |I_N'|)])$$

During measurement derivation, the internal state and the size of the corresponding enclave will be retrieved from the untrusted storage and authenticated within the enclave. In this way, the memory overhead becomes constant, as the common part stores only a hash value. The time cost for measurement derivation will also become constant as only a constant size of I_{common} is required to be updated to the measurement.

However, since the pre-measurement array is stored in untrusted storage, the overhead for its retrieval and integrity verification might still have a linear time complexity. To address this, the Merkle tree structure [22] could be adopted to organize the pre-measurement array outside the enclave memory (only the root hash of the Merkle tree is stored within I_{common}) for efficient retrieval and verification, achieving a logarithmic time complexity instead of a linear time complexity.

7.2 Extensions to Other TEEs

Remote attestation is a general concept in trusted computing. While this paper provides only one instantiation with Intel SGX, the proposed solution, MAGE, can be extended to other TEEs that use measurement-based identity to support mutual attestation. For example, AMD's Secure Encrypted Virtualization (SEV) is a TEE solution that encrypts the memory of virtual machines (VM) without a trusted hypervisor [18]. Remote attestation in SEV enables a guest owner to verify the initial integrity and authenticity of a guest VM launched on an SEV platform. The current generation of SEV, which is called SEV-SNP (Secure Nested Paging), allows the guest VM to request attestation reports at any time while the previous versions, *i.e.*, SEV and SEV-ES (Encrypted State), only support attestation during guest launch. The attestation report contains the guest VM's measurement and can be used by a remote user or another guest VM to verify its identity. While ARM TrustZone does not provide integrity measurement inherently, Zhao *et al.* proposed a software-based approach to provide secure enclaves using TEE such as ARM TrustZone [35]. The proposed scheme also includes a measurement-based identity. Hence, these TEEs could be integrated with MAGE to enable mutual attestation without TTPs. In addition, MAGE could be extended to work on TEEs that adopt more complicated measurement-calculation mechanisms. We describe one extension of MAGE for TEEs that leverage Merkle trees to compute measurements in Appendix A.

Further, MAGE can be extended for different types of TEEs to mutually attest each other. This could benefit applications that integrate different types of TEEs. For example, a privacy-preserving pandemic tracking system could be possible when mobile devices with ARM TrustZone are used to collect and transmit users' trajectories to cloud platforms with Intel SGX via secure channels established through mutual attestation. The collected trajectories could be monitored and analyzed privately within enclaves, and notifications would be returned to those affected mobile users. We expect mutual attestation between SGX and TrustZone would enable many other interesting use cases with cloud/client and edge/client computing models.

7.3 Supporting Enclave Updates

The current design of MAGE does not support enclave updates. If the content of any of the enclaves is changed, to continue the use of MAGE, all other enclaves need to be updated to reflect the change before these enclaves are re-deployed in the system. Hence, MAGE does not fit a system which is developed by multiple developers and requires frequent updates. On the other hand, when these enclaves are developed by the same developer, *e.g.*, in the cases of modularized applications and distributed applications, enclave updates could be fulfilled by the same developer directly. When these enclaves are from mutually distrusting parties, the lack of support of enclave updates in MAGE is intended. This is because updates of enclave code change not only the identity of the enclave but also the trustworthiness of its behavior. Therefore, a new version of an enclave should be inspected again for its trustworthiness. In other words, the trust relationship between these enclaves should be re-evaluated if one has been updated. Such enclave applications are similar to decentralized applications (dApps) built atop smart contracts, which are also difficult to patch once deployed. Therefore, solutions for dApps might also work for these enclave applications. We leave the investigation of facilitating enclave updates to future work.

7.4 Supporting Private Code

This paper assumes a trust model where an enclave is trusted by its identity, *i.e.*, its initial code and data, which implicates that its initial code and data should be publicly available and verifiable. This is undesired for enclave code and data with intellectual property rights, such as machine learning inference models. It is preferred to keep the content private. On the other hand, integrating such private enclaves into an application requires trust in their owners/developers. Given that the MAGE design for measurement-based identity requires only the pre-measurements and the sizes of these enclaves' the specific parts to function, the design could still apply for such a hybrid trust model: enclaves with intellectual property rights provide only pre-measurements and the sizes of their specific parts while the others additionally provide initial code and data.

8 Related Work

The work that is most related to ours is presented by Greveler *et al.* [12]. The authors proposed a protocol for two identical Trusted Platform Modules (TPM) to mutually attest each other for system cloning. The two identical TPMs generate the same value of the platform configuration register (PCR), which is the cryptographic hash of the software loaded into the TPM, having the same functionality as the measurement for TEEs. However, this protocol only works when both entities have the same identity, *e.g.*, PCR or measurement so that each entity could simply use its own measurement for verification. In contrast, our scheme enables enclaves with different measurements to mutually attest each other, enabling applications beyond system cloning. Shepherd *et al.* proposed a Bi-directional Trust Protocol (BTP) for establishing mutually trusted channels between two TEEs [25]. But BTP assumed that both TEEs know the identity of the other, while our work answers how this assumption could be realized.

Apache Teaclave, an open-sourced universal secure computing platform, addresses the mutual attestation problem by relying on third-party auditors [28]. Enclaves will be audited and signed by these auditors. The public keys of the auditors are hardcoded into these enclaves to support mutual attestation. These auditors are trusted by all involved enclaves and act as TTPs. On the contrary, MAGE tackles the mutual attestation problem without TTPs.

Also related to our work is a line of research on enclave migration. Park *et al.* was the first to address this problem by proposing a new SGX hardware instruction to be used to produce a live migration key between two SGX platforms for secure transfer of enclave content [23]. Gu *et al.* proposed a software-based solution by augmenting enclaves with a thread that could run remote attestation to establish a secure channel with the thread within another identical enclave, and then perform state transfer [13]. Alder *et al.* proposed an approach to migrate the persistent states of enclaves, *e.g.*, sealed data, which is outside of the enclave memory [1]. And Soriente *et al.* designed ReplicaTEE for seamless replication of enclaves in clouds [27]. While all these designs address secret migration between enclaves with the same measurement, our technique could complement them by enabling secret migration between enclaves with different measurements.

9 Conclusion

In this paper, we study techniques for a group of enclaves to mutually attest each other without trusted third parties. The main contribution of this paper is the mutual identity derivation mechanism that enables enclaves to derive other trusted enclaves' identities during runtime. We implement the proposed mechanism based on Intel SGX SDK and evaluate the performance. We demonstrate through a case study that this technique could facilitate new applications that require mutual trust for interaction and collaboration.

Acknowledgments

We are grateful to Jethro Beekman and Alexandra Dmitrienko for valuable feedback in shepherding our paper, as well as other reviewers for their insightful comments. The work was supported in part by the National Natural Science Foundation of China under Grant No. 62102254 and Shanghai Pujiang Program under Grant No. 21PJ1404900.

References

[1] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N. Asokan. Migrating SGX enclaves with persistent state. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 195–206, 2018.

[2] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Secure hash standard (shs). *Federal Information Processing Standards (FIPS) Publications (PUBS)*, Aug 2015.

[3] Jethro G. Beekman, John L. Manferdelli, and David Wagner. Attestation transparency: Building secure internet services for legacy clients. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 687–698, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1213–1227, Baltimore, MD, August 2018. USENIX Association.

[5] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[6] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. OPERA: Open remote attestation for Intel's secure enclaves. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2317–2331, New York, NY, USA, 2019. Association for Computing Machinery.

[7] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 185–200, 2019.

[8] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. Fastkitten: Practical smart contracts on bitcoin. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 801–818, Santa Clara, CA, August 2019. USENIX Association.

[9] Carl Ellison and Bruce Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Comput Secur J*, 16(1):1–7, 2000.

[10] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 275–294. USENIX Association, July 2021.

[11] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Ulrich Greveler, Benjamin Justus, and Dennis Löhr. Mutual remote attestation: Enabling system cloning for TPM based platforms. In *Security and Trust Management*, pages 193–206, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] 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 *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 225–236, 2017.

[14] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 299–312, Santa Clara, CA, July 2017. USENIX Association.

[15] Intel. Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation. https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf, 2018.

[16] Intel. Intel SGX SDK. https://github.com/intel/linux-sgx, 2019.

[17] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. Technical report, Intel, Tech. Rep, 2016.

[18] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. White paper, 2016. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[19] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, August 2017. USENIX Association.

[21] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, August 2017. USENIX Association.

[22] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, Berlin, Heidelberg, 1987. Springer-Verlag.

[23] J. Park, S. Park, J. Oh, and J. Won. Toward live migration of SGX-enabled virtual machines. In *2016 IEEE World Congress on Services (SERVICES)*, pages 111–112, 2016.

[24] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.

[25] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. Establishing mutually trusted channels for remote sensing devices with trusted execution environments. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, page 317–328, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Claudio Soriente, Ghassan Karame, Wenting Li, and Sergey Fedorov. ReplicaTEE: Enabling seamless replication of SGX enclaves in the cloud. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 158–171, 2019.

[28] Apache Teaclave. Mutual Attestation: Why and How, 2019. https://teaclave.apache.org/docs/mutual-attestation/.

[29] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1741–1758, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association.

[31] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2421–2434, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Wubing Wang, Yinqian Zhang, and Zhiqiang Lin. Time and order: Towards automatically identifying side-channel vulnerabilities in enclave binaries. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 443–457, Chaoyang District, Beijing, September 2019. USENIX Association.

[33] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 859–874, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[35] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. SecTEE: A software-based approach to secure enclave architecture using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1723–1740, New York, NY, USA, 2019. Association for Computing Machinery.

A Extending MAGE for TEEs that Leverage Merkle Trees to Compute Measurements

A Merkle tree is a binary tree with an assignment Φ that maps each node to a k-length bit-string: $n \longrightarrow \Phi(n) \in \{0,1\}^k$. Particularly, the value of a parent node is the cryptographic hash of its children nodes' values:

$$\Phi(n_{parent}) = \mathcal{H}\left(\Phi(n_{left})||\Phi(n_{right})\right)$$

And the value of a leaf node is the cryptographic hash of a data block.

A Merkle tree for an arbitrary-length message M, which is broken into fixed-length data blocks $M^1,\ldots,M^L$, can be built by firstly calculating the leaf nodes' values, *i.e.*, the cryptographic hashes of the data blocks $\mathcal{H}(M^1),\ldots,\mathcal{H}(M^L)$ and then calculating each parent node's value when its children nodes' values are ready. Let $\Psi(M)$ denote the root node of the built Merkle tree. The root hash of the built Merkle tree is its root node's value $\Phi(\Psi(M))$.

Consider TEEs that leverage Merkle trees to compute measurements. Particularly, an enclave `Encl` with content I could use the root hash as its identity $ID(I) = \Phi(\Psi(I))$. Given a group of N enclaves $\texttt{Encl}_\texttt{i}$ $(i = 1,\ldots,N)$, each of which has a specific part I'_i, one potential mechanism for mutual identity derivation is as follow:

- A common part generation $\mathcal{G}$ that calculates the root hashes of the specific parts of each enclave to form an array:

$$\begin{aligned} I_{\text{common}} =& \mathcal{G}(I'_1,\ldots,I'_N) \\ =& [\Phi(\Psi(I'_1)),\ldots,\Phi(\Psi(I'_N))] \end{aligned}$$

- A content builder function $\mathcal{C}$ that takes as input the specific part I'_i and the common part I_{common}, and produce the final content I_i such that the resulting Merkle tree has a root node $\Psi(I_i)$ with a left child node $\Psi(I'_i)$ and a right child node $\Psi(I_{\text{common}})$. So we have

$$\Phi(\Psi(I_i)) = \mathcal{H}\left(\Phi(\Psi(I'_i))||\Phi(\Psi(I_{\text{common}}))\right)$$

- An identity derivation function $\mathcal{F}$ that generates the identity of $\texttt{Encl}_\texttt{i}$ by retrieving the i-th entry from the common part, deriving $\Phi(\Psi(I_{\text{common}}))$ from I_{common} and completing the final root hash calculation:

$$\begin{aligned} &\mathcal{F}\left(I_{\text{common}}, i\right) \\ =&\mathcal{H}\left(I^i_{\text{common}}||\Phi(\Psi(I_{\text{common}})\right) \\ =&\mathcal{H}\left(\Phi(\Psi(I'_i))||\Phi(\Psi(I_{\text{common}}))\right) \\ =&\Phi(\Psi(I_i)) = ID(I_i), \forall i = 1,\ldots,N \end{aligned}$$