Intel SGX Remote Attestation is not sufficient

Yogesh Swami
yogesh.swami@gmail.com

July 30, 2017

Abstract

Intel SGX enclaves provide hardware enforced confidentiality and integrity guarantees for running pure computations (*i.e.*, OS-level side-effect-free code) in the cloud environment. In addition, SGX remote attestation enables enclaves to prove that a claimed enclave is indeed running inside a genuine SGX hardware and not some (adversary controlled) SGX simulator.

Since cryptographic protocols do not compose well [Cra96, Can00, HS11], especially when run concurrently, SGX remote attestation is only a necessary pre-condition for securely instantiating an enclave. In practice, one needs to analyze all the different interacting enclaves as a *single protocol* and make sure that no sub-computation of the protocol can be simulated outside of the enclave. In this paper we describe protocol design problems under (a) sequential-composition, (b) concurrent-composition, and (c) enclave state malleability that must be taken into account while designing new enclaves. We analyze Intel provided EPID [BL10] Provisioning and Quoting enclave [JSR$^+$16] and report our (largely positive) findings. We also provide details about how SGX uses EPID Group Signatures and report (largely negative) results about claimed anonymity guarantees.

1 Introduction

Intel SGX enclaves [MAB$^+$13, AGJS13] provide hardware enforced confidentiality and integrity guarantees for running pure computation (i.e., OS-level side-effect-free code) in the cloud environment. By limiting the application's Trusted Computing Base (TCB) to the CPU and CPU-Cache, SGX provides unprecedented confidentiality and integrity guarantees against malicious OS kernels and supervisor software. A popular design methodology—as evidenced by [BPH14, TAB$^+$14, ATG$^+$16]—for creating secure cloud applications is as follows:

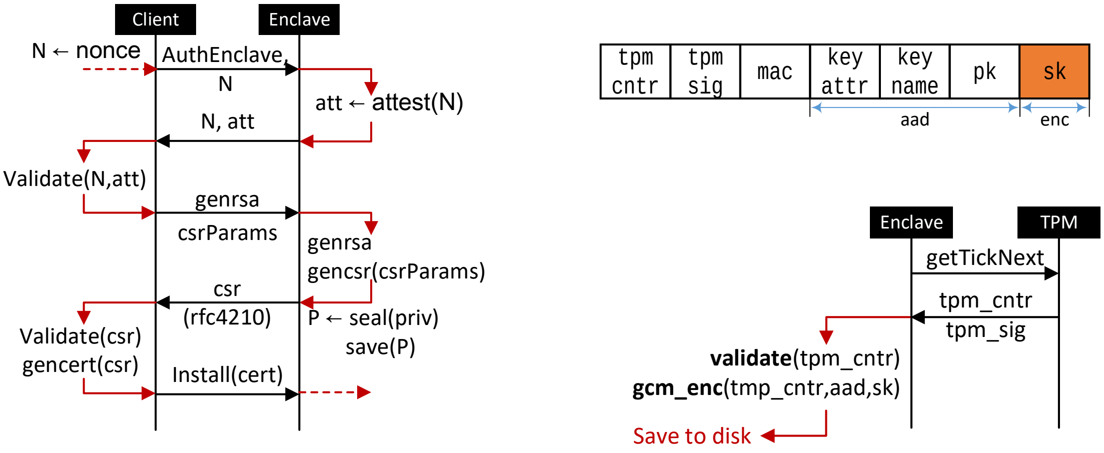
Step-1: First, define a remote-attestation mechanism to securely instantiate an enclave. Quite often, this step is not explicitly stated probably because a generic black-box attestation scheme—whatever that means—is expected to be sufficient.

Step-2: Then, largely independently of the remote-attestation mechanism, define the functionality that needs to be implemented inside the enclave. This step often involves composing different cryptographic as well as non-cryptographic protocols in ad-hoc ways to implement the desired algorithm. For example, the enclave may need to read encrypted keys from disk, compute a signature based on that key, create a new set of keys, log commitments of internal data, etc.

Step-3: Finally, define a "run-time workflow," where one first validates the remote-attestation result, and then runs the algorithm implemented by the enclave. This step often requires multiple interactions with various other entities such as other enclaves, untrusted host software, trusted remote client software, and other cryptographic devices such as TPMs.

It's hard to argue against the simplicity and ease of implementation of such a modular software design. However, it's well known [Can00, Cra96] that unless a protocol is designed for "Universal Composition" (UC)—where, the real-world behavior and the ideal-world definition (function) of a protocol are computationally indistinguishable *for every* adversary controlled environment—it's unlikely that arbitrary composition of such protocols will be secure. On the other hand, proving results in the UC-framework is rather difficult. In this paper we propose a framework for analyzing SGX enclaves that's a compromise between a full UC-based analysis and completely ad-hoc composition. Before describing the framework, we illustrate the problem associated with the protocol composition with two real-world examples.

To set the stage, a cloud service provider wanted to migrate its new clients from Amazon CloudHSM to an SGX enclave. The protocol for interacting with the enclave was based on HTTP Request/Response framework, where different operations (such as KeyGen), were sent as a command, and the enclave would execute and return a response (including explicit error codes) back to the remote caller. Important use-case for the enclave were to support (a) local key generation, (b) storing the public/private key on disk with an AEAD scheme that would allow *fast* key look-up, and (c) creating Certificate Signing Requests (CSR) from the enclave using challenge-response protocol [MKFA05, §5.2.8.3], among other things. Figure 1a describes one execution path of the protocol.



(a) Command execution for KeyGen with Cert

(b) Message format and seal protocol

Figure 1: Example of a flawed key-management enclave. The command execution protocol ascertains the authenticity of the enclave by validating the EPID signature on a randomly generated 256-bit nonce, followed by executing an arbitrary mix of commands as required by the use-case. Long-term keys are stored as GCM-encrypted AEAD blobs. The nonce (a 32-bit counter zero-padded on the left to 96-bits) for each GCM record is stored in TPM. The the TPM returns a signature on the nonce—along with some additional data—to disable roll-back of TPM "ticks." The enclave validates the TPM's signature before using the nonce for sealing.

This seemingly secure protocol is, in fact, not secure at all. Notice that the remote attestation in Figure 1a does not prevent a malicious cloud service provider from first faithfully responding to remote attestation queries, but then emulate the rest of the protocol (including KeyGen and CSR) outside of the enclave. While this is obvious in this simplified example, in a more complicated scenario, where multiple enclaves need to interact with each other, it might not be obvious if certain sub-components of the protocol can be simulated outside. Even though the entire enclave is *sequentially composed* from provably-secure protocols, the combined protocol is completely insecure!

Second, consider the seal protocol. Here each record (see Figure 1b) is GCM-encrypted using a nonce generated and signed by the TPM. However, consider a cloud service provider who instantiates two copies of the same enclave and *concurrently* executes KeyGen using the same TPM signed counter.

In this case, each enclave will generate two different keys in response to KeyGen. However, since the two concurrent instances will each correctly verify the signature (the two enclaves are identical), each will end up using the same nonce with different underlying data! As is the case with all counter modes, reusing a nonce can completely destroy the security of the system[1]. Note that this is not a flaw in GCM or in the way the TPM is used[2], rather, it's a case where an otherwise secure protocol is insecure under concurrent composition.

While these examples describe a totally broken scheme, in practice sequential and concurrent composition may not completely break the system as above. Rather it might just weaken the *bounds* of the entire protocol making it easy for further crypt-analysis or partial simulation of the enclave's protocol. For example, consider a scheme that consists of to two protocols π_1 and π_2, where the adversary needs 2^{t_1} and 2^{t_2} oracle queries to break π_1 and π_2 respectively. However, its possible that when composed sequentially as $(\pi_1 \circ \pi_2)$ or $(\pi_2 \circ \pi_1)$ the number of queries needed to break the composed protocol is smaller than $2^{\min\{t_1,t_2\}}$. In fact, since protocol composition rarely commutes, even different order of composition might result in very different bounds[3].

To summarize, *an enclave is a protocol* composed of several sub-protocols. In order for the enclave to be secure, it's essential that sequential and concurrent composition of sub-protocols remain secure. The rest of this document is organized as follows. §2 describes the abstract computational model of SGX that's better suited for security analysis. §3 describes pitfalls of sequential, concurrent, and parallel composition of cryptographic protocols and describes ways in which an enclave can be abused by a malicious cloud service provider. §4 describes Intel's remote attestation framework, and describes in detail the SGX remote attestation mechanism.

2 SGX Computational Model

Intel documentation [Int16a] provides excellent low-level details about the SGX instructions. This section provides an abstract computational model of SGX which is better suited for security analysis.

Abstractly, an SGX enclave can be thought of as a black-box that's capable of running any arbitrary algorithm. The black-box (enclave) can communicate with the outside world, called the environment, in three different ways:

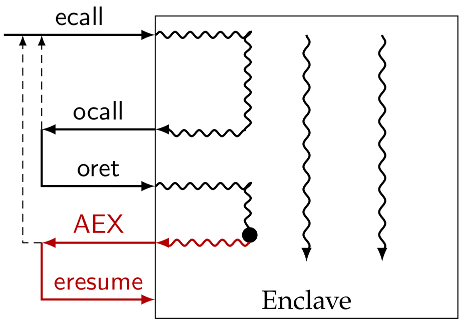


Figure 2: SGX Computational Model.

ecall: The environment can invoke a pre-defined function inside the enclave by passing input parameters and returning internal state of the enclave as results. Such invocations from the environment to the

---

[1] In the present case, since the underlying data is uniformly distributed, at least for AES or ECDSA keys, such a concurrent composition might not be harmful. However, if there is even a small bias in the random number generator, it might be possible to build a distinguisher from the xor of cipher-text data.

[2] When using TPMs with SGX enclaves, it's important that both the TPM and the enclave mutually authenticate each other. Failure to do so can lead to replay attacks where the adversary swaps the motherboard and in doing so resets the TPM counter. In the present case, however, even mutually authenticated TPM counter might not be secure under concurrent composition.

[3] Readers familiar with encrypt-then-mac vs. mac-then-encrypt debate should require no further explanation.

enclave are referred to as ecall. The parameter values passed from the environment to the enclave are either copied or directly shared with the enclave. An ecall can terminate in one of the three ways: (a) by returning normally as a function from the enclave, (b) by making an explicit ocall, or (c) as the result of an interrupt or exception. An ecall cannot halt all by itself.

SGX also supports multi-threading, and it's possible for the environment to run the same ecall in different threads. However, once an ecall has acquired the thread, future attempts to reuse that same thread will result in error. Furthermore, the number of threads that an enclave can support is pre-determined by the enclave signer, and cannot be altered at runtime.

ocall: While an enclave is executing (because of some previous ecall), it can make ocalls to pre-designated functions in the environment. Unlike an ecall, an ocall cannot directly share the internal enclave state with the environment, and must—directly or indirectly—copy the parameters into the environment before making an ocall.

An interesting characteristic of an ocall is that the environment is not required to return back to the enclave at the end of the ocall (see Figure 2). Since the behavior of pre-designated functions in the environment are controlled by the adversary, one should not expect the environment to follow the protocol that enclave author had envisioned. In particular, it's possible to create a chain of ecalls and ocalls such that the adversary can perform operations on the (global) internal state of the enclave. We call such adversarial manipulation of internal enclave state as *enclave malleability*.

Asynchronous Exit: In addition to an ocall, the processor can exit from an enclave due to an interrupt or exception. Such enclave exiting events are called Asynchronous Exit Events, or AEX. Unlike an ocall, an AEX can transfer control from the enclave to the environment at arbitrary (possibly adversary controlled) points inside the enclave. Like ocalls, an AEX can either by resumed from where the enclave left off, or the environment can invoke another ecall (either within the same thread or a different thread).

Since an adversary can create multiple running copies of an enclave and selectively interrupt each enclave to cause an AEX, it can be used as a means to "rewind" the internal state of the enclave. Given that proof-of-knowledge [BG93] protocols fundamentally have a *knowledge-extractor* based on rewinding, an enclave must ensure that it does not leak secrets when interrupted by an AEX.

2.1 Enclave Creation

An enclave is generated as a dynamically shared library using standard compiler tools. In addition, the entity creating the enclave must also decide up-front on the following information:

Attributes: The attributes of an enclave act as an access control mechanism that is enforced by the hardware. For example, certain high privilege keys, such as Launch Key and Provisioning Key, cannot be made accessible to all the enclaves as it would compromise the security of entire SGX ecosystem. In order to gain access to these keys, an enclave author must explicitly request for these attributes at compile/sign time. During enclave launch-time, the Launch Enclave, based on policy decisions, decides whether to grant or reject requests based on these attributes.

Stack size: The enclave author must estimate the size of the stack needed by the enclave and set its value at enclave creation time. Once an enclave is instantiated, this value cannot be changed.

Heap size: Like the stack size, the heap-size of the enclave is also fixed at enclave creation time. In SGXv2, this value can be changed post-instantiation.

Thread count: An enclave must also decide upon the number of threads that can run concurrently. As pointed out in §2, concurrency can have a dramatically negative impact on the security of the certain protocols, and one must not select this parameter just on the basis of performance requirements, but also on the basis of security concerns.

Software version: SGX provides elaborate software-upgrade and life-cycle management facilities and allows software vendors to make use of these features.

Based on these parameters, the enclave signing tool creates a virtual memory layout of the enclave and computes a hash of the entire memory layout (including the stack, heap, thread control structure, etc.) See [Int16a] for details about how the hash is computed. This hash, called mrenclave, is used as the unique identifier for the enclave.

In addition to mrenclave, the software vendor must also sign the enclave using a RSA-3072 key. The hash of the RSA Public-Key is called mrsigner. As described in [BG17], the purpose of the signature is to provide an unforgeable identity—a *surname* based lineage—to a set of enclaves based on the vendor.

It should be noted that the mrenclave of an enclave doesn't change even when the signing key is changed. This is significant when validating attestation or deriving keys based on mrenclave.

2.2 Enclave instantiation and access control

A properly signed enclave can be instantiated on any Intel SGX Processor—subject to access control restrictions enforced by Launch Enclave. Before an enclave can be instantiated on an SGX capable processor, it must first get an authorization token, called Launch Token, from Intel provided Launch Enclave. The Launch Enclave uses a combination of mrenclave, mrsigner, the attributes of the enclave and a *white-list signed by Intel* to decide whether to grant Launch Token or not. Once an enclave obtains a Launch Token, it can continue using it indefinitely—even when the policies of the Launch Enclave might get updated later on and deny access to Launch Token for that enclave!

2.3 SGX Platform Keys

As described in [JSR$^+$16], each SGX capable processor contains two statistically independent base keys: Root Provisioning Key and Root Seal Key. The Root Provisioning Key is used as the *root-of-trust* between the CPU and Intel Attestation Services (IAS) [Int17]. Intel retains a copy of this key at the time of manufacturing and uses it to establish the trustworthiness of the processor during EPID join process. Intel claims that Root Seal Key is not retained. However, it's not clear whether this key is generated inside the processor via oracle access (*i.e.*, in such a way that CPU generates the key all by itself using it's own internal random numbers or with PUFs), or whether the key is first generated outside the processor, then injected into the CPU, and finally the outside references are destroyed. Unless these keys are generated via oracle access, one should consider Root Seal Key to be known to Intel.

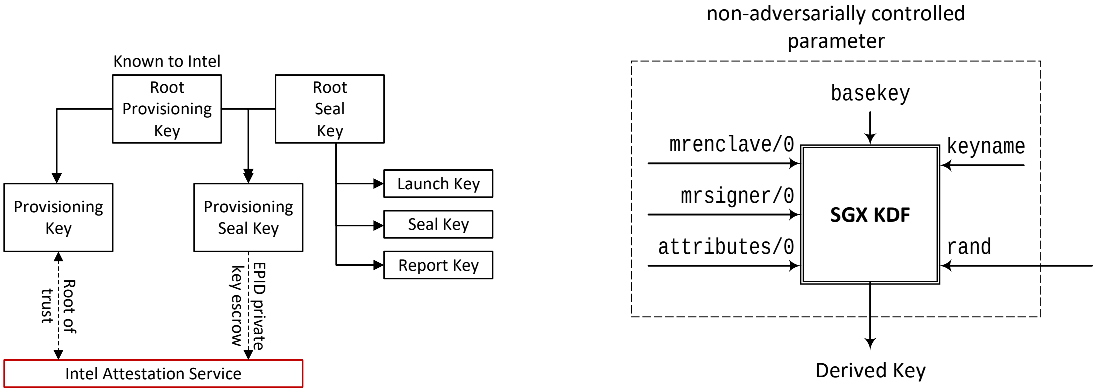
By design, an application software does not have raw access to the base keys. However, an application can access *named* keys that are derived from these two base keys (see Figure 3). The key derivation function allows enclave authors to specify policies on how to derive enclave specific keys from base keys. These policies include using the CPU resident (*i.e.*, trusted) values of mrenclave, mrsigner and/or attributes of the enclave.

An implication of this design is that enclaves cannot derive keys that might belong to a different enclave's mrenclave or mrsigner. Furthermore, when key derivation policy allows skipping specific fields (such as mrenclave), a default value of all-zeros is automatically used. Therefore, even when such un-specialized keys are available, it's not possible to derive specialized keys from them.

Remark: While it's possible to access some keys (e.g., Seal Key) that have neither been specialized with mrenclave nor mrsigner, such keys *should not* be used: An attacker can create a standalone enclave that also does not specialize the key in anyway and come to the same value. We consider this to be a design flaw in SGX.

The following list describes all the named keys and their intended usage:

Provisioning Key: This key is derived from Root Provisioning Key and is used as a version dependent root-of-trust between Intel Attestation Service and SGX capable processor. Since admitting a non-SGX processor to the the EPID group of SGX processors will completely compromise remote attestation for all CPUs, extreme care must be taken in granting access to Provisioning Key.



(a) The Provisioning Key acts as a root-of-trust between SGX capable CPU and Intel Attestation Service. Provisioning Seal Key is used for EPID private key escrow.

(b) SGX Key derivation function. Only parameters outside the dotted line can be chosen maliciously. Key derivation uses all-zeros for mrenclave, mrsigner, and attributes if key policy doesn't specify which ones to use. See [Int16a, §38.17] for additional details.

Figure 3: SGX Platform and Named Key.

Currently, the Launch Enclave only grants access to this key if the enclave was signed by Intel. (Intel's mrsigner is hard-coded into the Launch Enclave and this policy enforcement cannot be circumvented.)

Provisioning Seal Key: This key is derived jointly from Root Provisioning Key and Root Seal Key. During the EPID join process, the EPID private-key for each platform is encrypted with this key and uploaded to Intel Attestation Service. (See §4.2 for details about EPID join process.)

Note that the EPID private-key could not just be encrypted with Provisioning Key as that would destroy the EPID's blinded-join protocol. Conversely, the EPID private-key cannot be encrypted just with Seal Key as that might allow non-privileged enclaves to have access to EPID private key.

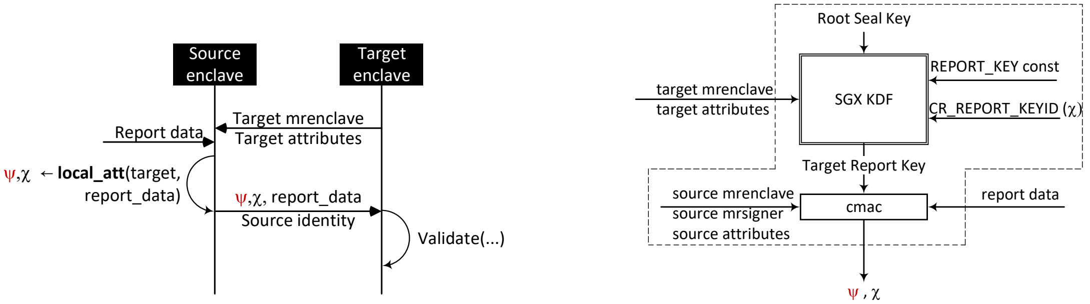
In spite of this design choice, given the uncertainty about how the Root Seal Key is generated, one should assume that Intel knows the EPID private key for each platform.

Launch Key: This key is derived from Root Seal Key and is used by Launch Enclave to create authorization tokens (EINITTOKEN). Recall that each non-Intel enclave must obtain this authorization token before the CPU can instantiate the enclave. Only a specific mrsigner—whose corresponding private-keys are only known to Intel—can access the Launch Key. In SGXv2, the mrsigner for Launch Enclave can be changed programmatically [Int16a, §39.1.4], but it's not clear how Intel intends to enforce access control restrictions on Provisioning Key.

Remark: It's unclear why Launch Key needs to be derived from a long-term secret (Root Seal Key). An ephemeral Launch Key generated at the processor boot-up would not only be more secure, but also enable better policy enforcement by Launch Enclave.

Seal Key: This key is derived from Root Seal Key and used for encrypting data specifically for a given CPU. As we remarked earlier, one must not use un-specialized Seal Key—either for encryption or authentication—as that would completely compromise the security of that enclave.

Report Key: This key is derived from Root Seal Key and used for Local Attestation (see §2.4 for detailed information on Local Attestation and how Report Key is used).



(a) Local attestation message flow.

(b) Local attestation computation. Parameters outside the dotted line can be adversarially selected.

Figure 4: Local attestation computation and message flow

2.4 Local Attestation

The process of local attestation allows a source enclave (source-enclave) to prove to a target enclave (target-enclave)—running locally on the same platform—that the source-enclave is indeed running on a genuine Intel SGX platform (see Figure 4a). In addition, the source-enclave can optionally use 512-bits of additional data (e.g., hash of public-key), called report-data, to claim knowledge of certain bit-string.

The process of local-attestation involves computing CMAC [ISLP06] on the source-enclave's identity (i.e., mrenclave, mrsigner, etc.) using the target-enclave's Report Key. However, as pointed out in §2.3, the source-enclave cannot directly access target-enclave's Report Key. SGX solves this problem by providing *oracle access* to target-enclave's Report Key via EREPORT instruction [Int16a, §14.4.1].

To compute local attestation, the source-enclave obtains the mrenclave and attributes of the target-enclave through some out-of-band mechanism (which might be adversarial). Based on target-enclave's mrenclave, the EREPORT instruction internally derives the target-enclave's Report Key and computes CMAC on source-enclave's identity: mrenclave, mrsigner, and attributes. To prevent malicious enclaves from forging their identity, the CPU's internal trusted cache is used for computing source-enclave's identity. During target-enclave's key derivation, the EREPORT instruction uses a boot-time random number called CR_REPORT_KEYID, which is also returned in addition to CMAC.

Since the target enclave can directly access its Report Key, the verification involves manually fetching the Report Key and verifying the CMAC on report body. The report body includes the identity (mrsigner, mrenclave, attributes) of the source-enclave, and optionally 512-bits or report data.

Remark: It's unclear why local attestation needs to be tied to a long-term secret (Report Key) of the processor.

3 Enclave Malleability and Knowledge Extractors

Given the computational model of SGX, we describe certain pitfalls in enclave design that might inadvertently make the enclave malleable, or open door for building knowledge-extractors [BG93].

3.1 Enclave malleability

As described in §2, an application can exit an enclave either (a) as a function return from an ecall (b) as an ocall or (c) as an AEX. Since it's not required for an ocall or AEX to return back to the enclave from the state it left off, it's possible for a malicious environment to make unexpected ecalls to alter the internal state of the enclave. Enclaves whose global internal state can be influenced by an attacker by not following the expected protocol are called *malleable enclaves* in this document.

To better understand enclave malleability, consider the following example: The US government wants to use an SGX enclave to implement 2-man rule for launching nuclear missiles. The 2-man rule requires that at least two *different* members (generals) of the armed forces to authorize the launch a nuclear missile.

Listing 1 describes one way to implement this. Essentially, the enclave keeps a list of generals, their public-keys, and their individual authorization state in a global variable `GENERALS`. In addition, the enclave keeps the number of distinct generals who have authorized the launch in a global variable `auth_count`. Since different generals might be authorizing the launch at different times, the enclave allows each general to authorize a launch individually by signing the concatenation of general's name and some auxiliary data.

```
 1 /* count of generals who have authorized launch. */
 2 static int auth_count = 0;
 3
 4 /* hardcoded list of generals and their PKs */
 5 struct general_info{
 6   char general_name[256];
 7   const sgx_ec256_public_t general_pub;
 8   bool has_authorized; // initialized to false
 9 }GENERALS[] = { ... };
10
11 /* ecall made by each general with a sig on name + aux data */
12 int auth_and_launch(const char* const general_name,
13                     const sgx_ec256_signature_t* sig){
14
15   struct general_info* valid_general =
16           validate_general(general_name, sig);
17
18   if(!valid_general){ return INVALID_GENERAL; }
19
20   if(!valid_general->has_authorized){
21     auth_count++; // AEX here will be devastating!
22     valid_general->has_authorized = true;
23   }else{
24     return GENERAL_ALREADY_AUTHORIZED_ACTION; // replay
25   }
26
27   if(auth_count >= 2){
28     return nuke_the_kashbah(location);
29   }
30
31   return PENDING_AUTHORIZATION;
32 }
```

Listing 1: An enclave suseptible to state malleability

Can this enclave be exploited to launch a missile with *just one* authorization, say $\langle g_1, \sigma_1 \rangle$? Surprisingly, the answer is yes! Here is how:

1. The attacker first feeds $\langle g_1, \sigma_1 \rangle$ to `auth_and_launch` function with the intent of causing an AEX between lines 21 and 22. Since the attacker can artificially cause an interrupt and also instantiate multiple copies of the enclave in parallel, given a polynomial number of trials (in program

length), the attacker can cause an AEX between line 21 and 22 w.h.p. Note that at the time of a successful AEX between line 21 and 22, the `auth_count` and `has_authorized` variables will be in an inconsistent state where the `has_authorized` would still be `false` and another ecall to `auth_and_launch` will successfully update the `auth_count` variable.

2. After the enclave has been interrupted by AEX and the processor is ready to resume, the attacker instead of resuming, makes an ecall to `auth_and_launch` again with the same old parameters $\langle g_1, \sigma_1 \rangle$. Since the first ecall had incremented the counter, but left the authorization state inconsistent, the second ecall will once again increment `auth_count`, ultimately leading to a nuclear attack!
3. While not applicable in this case, in some cases it might be necessary for an attacker to resume the first ecall after the second one has completed. Since the enclave preserves the stack before making an AEX, resuming the first ecall tantamount to executing ERESUME assembly instruction.

It should be emphasized that the problem of state-malleability is broader in scope than the race condition described above. For example, one can use malleability to induce an error which turns the enclave into an oracle. It should also be emphasized that enclaves should return error codes with security consideration in mind.

3.2 Enclave rewinding and knowledge-extractors

Zero-Knowledge Proof-of-Knowledge (ZKPK) protocols, by definition have in-built knowledge-extractor [BG93, Mau09]. The knowledge-extractor is designed by giving a simulator the capability to "rewind" the prover's state to arbitrary point in it's execution. Since SGX enclaves can be interrupted by an AEX, it's important that a malicious environment is not able to rewind the enclave in such a way that it inadvertently reveals the secret-key.

Consider the three-move—commit, challenge, blinded-reveal—Σ-protocols [Dam] that are the most efficient and widely-used ZKPKs protocol in practice. Normally, one designs Σ-protocols *with interaction* between a prover and a verifier in mind, and then uses Fiat-Shamir [FS87] heuristic[4] to convert it into a useful non-interactive use-case such as a signature scheme. Most of these protocols just require the prover to respond to two challenge message for a given commitment message to reveal the secret.

If an enclave is not implemented appropriately, one can induce an artificial AEX right after the commitment phase, and call the enclave with different messages in possibly different threads to generate two responses to the same commitment message. Note that AEX in conjunction with multi-threading opens doors for a limited form of enclave rewinding and presents a larger attack surface than AEX alone. Unless, an enclave requires multi-threading, it's wise to set the number of possible threads to the bare minimum.

4 SGX remote attestation

SGX is an example of a hardware/software co-design of a cryptographic platform. A common concern in the design of such systems is to ensure that an adversary is not able to switch the hardware with a software simulator (such as QEMU [Bel05, JDK^{+}16]) of the hardware. Since an Universal Turing Machine can simulate any piece of computing hardware, unless there's an inbuilt asymmetry between what the software "knows" and what the hardware knows, it's impossible to prevent software simulator attacks in such systems. On the other hand, each independent piece of software must somehow have raw or oracle access to the hardware's secret so that it can prove to remote parties that it's running on a real hardware. The essence of any remote-attestation scheme in such systems is to address these two conflicting requirements. *Note*: Limiting access to raw hardware keys via an oracle is not sufficient

---

[4] In the Fiat-Shamir heuristic, the prover also *pretends* to be an honest verifier and generates the challenge string via a random oracle, based on publicly known fields of the protocol (such as the commitment value, user's input message, etc.).

to thwart simulator based attacks. An attacker can run the hardware simulator on a real hardware, gain access to the hardware-secret via the oracle, and then impersonate as the real hardware.

In case of Intel SGX, the question of knowledge-asymmetry between hardware and software is answered by the Root Provisioning Key (see §2.3). The dilemma of both denying as well as granting access to this hardware secret is solved by a two-step process:

1. Intel has created a (set of) privileged enclaves—called Provisioning Enclave (PvE) and Provisioning Certification Enclave (PcE)—that have raw access to Provisioning Key and Provisioning Seal Key. The PvE and PcE use Provisioning Key and Provisioning Seal Key to bootstrap a new set of software-only credentials for a group-signature scheme called Enhanced Privacy ID (EPID) [BL10]. Since only Intel signed enclaves have access to Provisioning Key and Provisioning Seal Key no malicious simulator can access these keys.

2. Once a platform has been provisioned with EPID keys, another Intel signed enclave called Quoting Enclave (QE) is given raw access to EPID keys and made responsible for generating remote-attestation results on behalf of other—potentially malicious—enclaves.

The rest of this section is organized as follows: §4.1 provides an overview of EPID and how it's has been implemented by Intel. Since the official [BL10] paper leaves several details out (e.g., the Zero-Knowledge proof of inequality for signature based revocation), the goal of this section to fill in those gaps based on open source implementation of `epid-sdk` [Int16b]. §4.2 provides detailed information on how the Provisioning Enclave joins the SGX EPID group.

4.1 EPID Overview

In a standard signature scheme, such as ECDSA or RSA-PSS, each signer has a unique private/public key-pair. Given two message/signature pairs $\langle m_1, \sigma_1 \rangle$ and $\langle m_2, \sigma_2 \rangle$, an attacker in possession of N public-keys can easily determine if m_1 and m_2 were signed by the same private key or not. If such signatures are generated by physical devices, it can be used to track the signing device and thereby destroy the anonymity and privacy of the person using that device.

Group signatures were introduced by Chaum and Van Heyst [CH91] as a means to address this. Their idea was to create a signature-scheme where a single "group public-key," can verify messages signed by different private keys. In order to achieve this, a designated entity called Group Manager admits members to the group and grants *membership credentials* in such a way that a single public-key can verify messages signed by different private keys. In addition to existential-unforgeability required for signatures, a group signature also requires *traceability* and *non-frameability* to keep members accountable. Alternatively, as is the case with EPID, a group membership revocation mechanism is required to deal with fraudulent members.

The literature on group signature schemes is huge, both for formal models of its security as well as for different constructions using different computational assumptions [BMW03, BSZ05, BCC$^+$16, BBS04, FI05, ACJT00, CL04]. Among these, from a practical deployment perspective, Direct Anonymous Attestation [BCC04, CDL16] (DAA) is closest to EPID and also most widely deployed. We do not review these schemes any further in this paper.

At a high-level, EPID signatures have two distinct components: The first component, called the `BasicSignature`, is based on BBS+ (ordinary[5]) signature scheme [ASM06]. The second component is a pair of (algebraic) group elements per-signature, to facilitate signature based revocation (described in detail below). Because of space and time constraints, we intentionally leave out additional details about the `BasicSignature` as it's adequately described and proven secure in [BL10, ASM06, BBS04].

From a practical perspective, there are four entities in EPID:

---

[5] To clarify, BBS+ is an ordinary CCA2 secure signature scheme like RSA-PSS, but unlike PSS, it's secure in standard model under q-SDH assumption. BBS+ itself is derived from Boneh-Boyen-Shacham [BBS04] group-signature.

Issuer ($\mathcal{I}$): It's the entity that grants group membership credentials to its members. In case of SGX, the Intel Attestation Service acts as the Issuer. Its goal is to dynamically add new SGX Processors as they come on-line.

Revocation Manager ($\mathcal{R}$): It's the entity that decides who are the (known) offending members of the group. Unlike standard signature schemes, where revocation only includes the public-key of fraudulent signers, group signatures require a different approach. EPID has two forms of revocation:

`Priv-RL` : Private-key based revocation list. `Priv-RL` is a list of compromised *private-keys* known to Revocation Manager. EPID does not support full-anonymity in the sense of [BMW03][6], and putting a private key in `Priv-RL`, retroactively destroys the anonymity of the signer.

`Sig-RL` : Signature based revocation list. An EPID signature consists of a basic BBS+ [ASM06, BBS04] based signature along with two group elements $\langle B, B^f \rangle$ from a group where Discrete-Log is hard. B is called the basename and f is the EPID private-key. The `Sig-RL` consists of a list of $\langle B_i, B_i^{f_i} \rangle$ pairs from previously signed messages that the Revocation Manager believes to have been signed fraudulently. To sign a message, an honest signer must pick a random[7] B and prove in Zero-Knowledge that none of the $\langle B_i, B_i^{f_i} \rangle$ pairs in the `Sig-RL` could have been generated using the signer's f (essentially prove in Zero-Knowledge that none of the discrete logs in the list equal to f).

Remark: For any f, a single signature-revocation-pair $\langle B_i, B_i^{f} \rangle$ is sufficient to identify fraudulent signers, however, this does not mean that the length of `Sig-RL` is limited by the number of signers. A fraudulent signer can still keep signing messages, until the Revocation Manager caches up. At which point, the Revocation Manager will not be able to decide if a signature from fraudulent signer is already present in `Sig-RL`, and will therefore need to put all the suspected signatures in `Sig-RL`. Because of this, `Sig-RL` can become very large.

Since the EPID paper leaves out details about zero-knowledge proof of inequality, we point out that SGX implementation [Int16b] uses the scheme described in [CS03, §6].

In EPID, a signer needs to have access to the most up-to-date `Sig-RL` to generate a valid signature. This is fundamentally different from *verifier local revocation* (VLR) [BS04] where the signer never needs up-to-date revocation list to *generate* a valid signature (the verifier, of course, always needs up-to-date revocation list). Also, unlike verifier local revocation, EPID signatures are of variable length and even the same message signed with the same private-key can have different lengths depending upon the length of `Sig-RL`. It's surprising that such a variable-length signature scheme can still be anonymous!

Platforms ($\mathcal{P}$): Platforms in EPID are entities that are part of the signing group. In case of SGX, each SGX capable CPU SoC is a platform.

In SGX, the Provisioning Enclave is responsible for executing the blinded-join protocol and securely storing the member's group credentials to permanent storage. Once Provisioning Enclave has obtained its membership-credentials from Intel Attestation Service, it stores them on disk encrypted with Seal Key (derived with mrsigner of PvE). Only Intel Signed enclaves can access the EPID signing key.

Verifiers ($\mathcal{V}$): Any entity in possession of the group public-key is a verifier. In case of Intel Attestation Service, however, each signature is encrypted using an authenticated public-key in the Quoting

---

[6] In the anonymity game of [BMW03], the adversary gets the private key of all the members, and yet it cannot distinguish one signer from another based on signatures alone.

[7] A signer (or an adversary) may choose to use the same value of B for different signatures if they want signatures to be linked. In SGX, B is chosen from an Elliptic Curve group and computed by hashing a 256-bit number B' to a point on Elliptic Curve. The first 128-bits of B' is always set to Service Provider ID (SPID), even for unlinkable signatures.

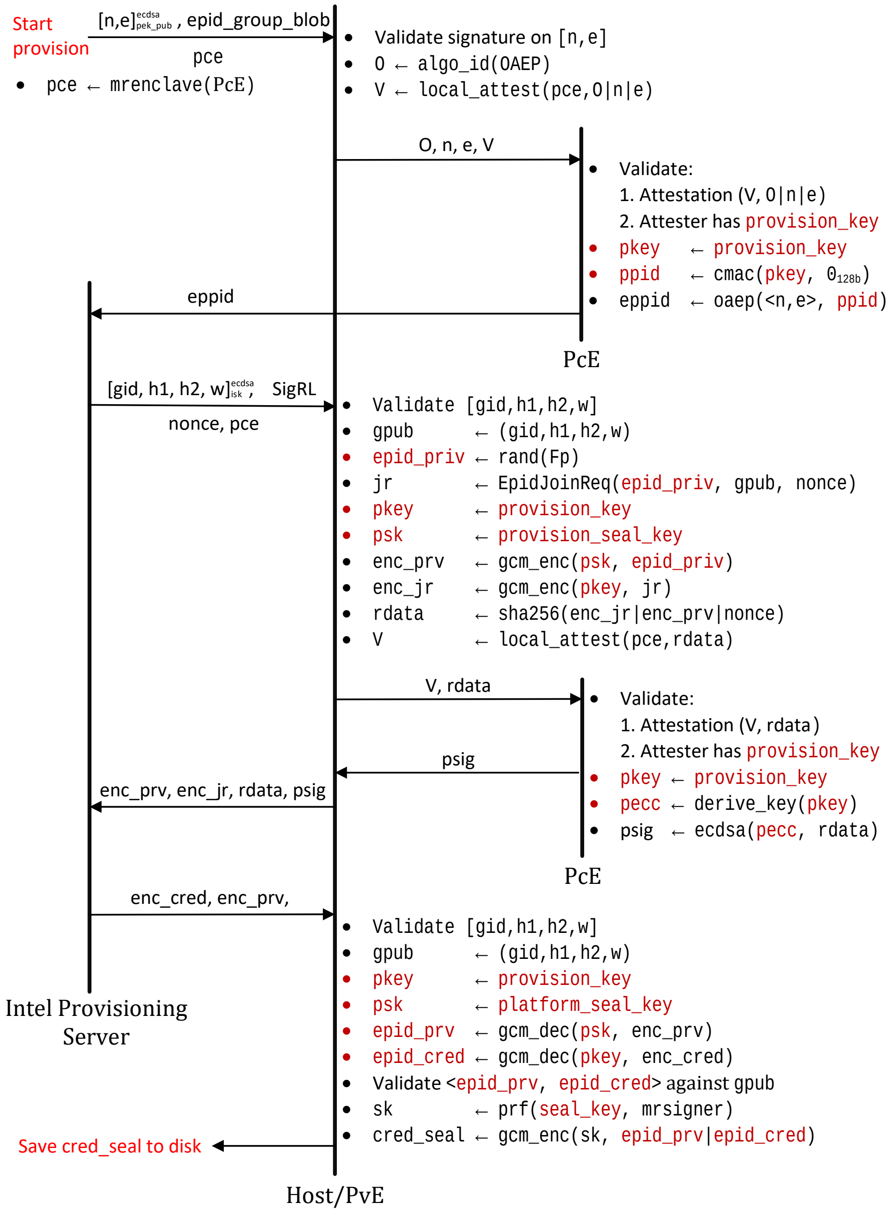


Figure 5: EPID Provisioning Protocol. The Platform side of provisioning is split between PvE and PcE enclaves. The EPID join request is encrypted using a key derived from Provisioning Key. Furthermore, an ECDSA signature is computed on the encrypted join request, using a key derived from Provisioning Key. This signature acts as a proof-of-knowledge of Root Provisioning Key for IAS to verify. The provisioning process also encrypts the EPID private key using Provisioning Seal Key and sends it as escrow data to IAS.

Enclave. Since only Intel Attestation Service can decrypt these signatures, only Intel Attestation Service can verify signatures!

Remark: Not having the ability to verify signatures locally means that one must trust Intel Attestation Service even to validate the signature. If Intel, for whatever reason, chooses to lie about the validity of a signature, it could be used to launch man-in-the-middle attacks on enclaves. Furthermore, each encrypted EPID signature in SGX also contains the service provider's ID (SPID), which can be used to track the number of times a service provider is interacting with SGX processor.

4.2 SGX EPID provisioning

In order to create new set of EPID credentials, the SGX capable processor must participate in the EPID Join process. When presented with a join request, the Intel Attestation Service must somehow ensure that the join request indeed came from an SGX processor; allowing non-SGX platforms to join SGX EPID group would render the entire remote attestation scheme useless. To make matters worse, under concurrent composition, the Zero-Knowledge Proof of Knowledge Protocol used in the EPID Join request is not secure, and the Intel Attestation Service must somehow prevent arbitrary interleaving of Join messages.

Intel has addressed these issues by creating two Intel signed enclaves called PvE and PcE[8]. Both these enclaves have access to the Provisioning Key, and the mrsigner for these enclaves is hard-coded in the Launch Enclave—preventing non-Intel enclaves from gaining access to the Provisioning Key.

Figure 5 describes the details about EPID provisioning.

5 Conclusion

This paper describes the pitfalls in designing SGX enclaves. In particular, it highlights issues with sequential and concurrent composition of protocols. In addition, we also describe issues with enclave state-malleability which must be taken into account when implementing new enclaves.

Based on these three criteria, we have analyzed Intel provided PvE, PcE, and QE enclaves and found them to be secure. On the other hand, we find following issues with current implementation of Intel provided enclaves:

- The EPID join process uses raw Provisioning Key and long-term platform identifiers (PPID) that are derived from this key. This essentially destroys the anonymity of the platform during the EPID join process. We note that in case of SGX, it's possible to build a Witness-Indistinguishable (WI) or Witness-Hiding (WH) join protocol that not only guarantees anonymity, but also might alleviate concerns about concurrent join.
- In its current implementation, the Quoting Enclave encrypts all EPID signatures that can only be decrypted and verified by the Intel Attestation Service. Not only does this destroy the anonymity of signers, but also allows Intel to facilitate man-in-the-middle attacks on enclaves, should Intel choose (or be compelled) to do so.

6 Acknowledgements

Many thanks to Mike Hamburg for his comments and feedback on this paper.

---

[8] While we not sure why the provisioning process was split into two separate enclave with the same set of privileges, we believe this is done to separate the enclave that directly interacts with network data (PvE) from the one that only signs (certifies) messages (PcE).

References

[ACJT00] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '00, pages 255–270, London, UK, UK, 2000. Springer-Verlag. `http://www.ics.uci.edu/~gts/paps/acjt00.pdf`.

[AGJS13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. *Workshop on Hardware and Architectural Support for Security and Privacy*, June 2013. `https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing`.

[ASM06] Man Ho Au, Willy Susilo, and Yi Mu. *Constant-Size Dynamic k-TAA*, pages 111–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. `http://eprint.iacr.org/2008/136`.

[ATG$^+$16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, GA, 2016. USENIX Association. `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov`.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. *Short Group Signatures*, pages 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. `https://www.iacr.org/archive/crypto2004/31520040/groupsigs.pdf`.

[BCC04] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 132–145, New York, NY, USA, 2004. ACM. `https://eprint.iacr.org/2004/205.pdf`.

[BCC$^+$16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, and Jens Groth. *Foundations of Fully Dynamic Group Signatures*, pages 117–136. Springer International Publishing, Cham, 2016. `http://eprint.iacr.org/2016/368`.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[BG93] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 390–420, London, UK, UK, 1993. Springer-Verlag. `http://dl.acm.org/citation.cfm?id=646757.759584`.

[BG17] Dan Boneh and Shay Gueron. Surnaming schemes, fast verification, and applications to sgx technology. *CT-RSA: RSA Conference Cryptographers' Track*, pages 149–164, February 2017.

[BL10] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing for hardware authentication and attestation. *IEEE International Conference on Social Computing / IEEE International Conference on Privacy, Security, Risk and Trust*, pages 768–775, August 2010. `https://eprint.iacr.org/2009/095.pdf`.

[BMW03] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: formal definition, simplified requirements and a construction based on trapdoor permutations. In Eli Biham, editor, *Advances in cryptology - EUROCRYPT 2003, proceedings of the internarional conference on the theory and application of cryptographic techniques*, volume 2656 of *Lecture Notes in Computer Science*, pages 614–629, Warsaw, Poland, May 2003. Springer-Verlag. `http://cseweb.ucsd.edu/~daniele/papers/BMW.pdf`.

[BPH14] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI14)*, pages 113–124, August 2014.

[BS04] Dan Boneh and Hovav Shacham. Group signatures with verifier-local revocation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 168–177, New York, NY, USA, 2004. ACM. `https://cseweb.ucsd.edu/~hovav/dist/preteripsistic.pdf`.

[BSZ05] Mihir Bellare, Haixia Shi, and Chong Zhang. *Foundations of Group Signatures: The Case of Dynamic Groups*, pages 136–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. `https://cseweb.ucsd.edu/~mihir/papers/bsz.pdf`.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/2000/067`.

[CDL16] Jan Camenisch, Manu Drijvers, and Anja Lehmann. *Universally Composable Direct Anonymous Attestation*, pages 234–264. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. `http://eprint.iacr.org/2015/1246`.

[CH91] David Chaum and Eugène Van Heyst. Group signatures. In *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'91, pages 257–265, Berlin, Heidelberg, 1991. Springer-Verlag.

[CL04] Jan Camenisch and Anna Lysyanskaya. *Signature Schemes and Anonymous Credentials from Bilinear Maps*, pages 56–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. `http://cs.brown.edu/~anna/papers/cl04.pdf`.

[Cra96] Ronald Cramer. *Modular design of secure, yet practical cryptographic protocols*. PhD thesis, University of Amsterdam, 1996.

[CS03] Jan Camenisch and Victor Shoup. *Practical Verifiable Encryption and Decryption of Discrete Logarithms*, pages 126–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[Dam] Ivan Damgård. On Σ protocols. `https://eprint.iacr.org/2010/284.pdf`.

[FI05] Jun Furukawa and Hideki Imai. *An Efficient Group Signature Scheme from Bilinear Maps*, pages 455–467. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. `http://dx.doi.org/10.1007/11506157_38`.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 186–194. Springer-Verlag, 1987.

[HS11] Dennis Hofheinz and Victor Shoup. Gnuc: A new universal composability framework. Cryptology ePrint Archive, Report 2011/303, 2011. `http://eprint.iacr.org/2011/303`.

[Int16a] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. December 2016. `https://software.intel.com/en-us/articles/intel-sdm`.

[Int16b] Intel Open Source (`https://01.org`). *EPID SDK R3*. 2016. `https://01.org/epid-sdk/downloads`.

[Int17] Intel Corporation. *Attestation Service for Intel® Software Guard Extensions: API Documentation*. 2017. `https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf`.

[ISLP06] Tetsu Iwata, Junhyuk Song, Jicheol Lee, and Radha Poovendran. The AES-CMAC Algorithm. RFC 4493, June 2006. `https://rfc-editor.org/rfc/rfc4493.txt`.

[JDK$^+$16] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. OpenSGX: An Open Platform for SGX Research. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2016.

[JSR$^+$16] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® software guard extensions: Epid provisioning and attestation services. March 2016. `https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services`.

[MAB$^+$13] Frank Mckeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isolated execution. *Workshop on Hardware and Architectural Support for Security and Privacy*, June 2013. `https://software.intel.com/en-us/articles/innovative-instructions-and-software-model-for-isolated-execution`.

[Mau09] Ueli Maurer. Unifying zero-knowledge proofs of knowledge, 2009. `ftp://ftp.inf.ethz.ch/pub/crypto/publications/Maurer09.pdf`.

[MKFA05] Tero Mononen, Tomi Kause, Stephen Farrell, and Dr. Carlisle Adams. Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP). RFC 4210, September 2005. `https://rfc-editor.org/rfc/rfc4210.txt`.

[TAB$^+$14] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. *In Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, pages 113–124, April 2014.