CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks

Fergus Dall[1], Gabrielle De Micheli[2], Thomas Eisenbarth[3,4], Daniel Genkin[2,5], Nadia Heninger[2], Ahmad Moghimi[4] and Yuval Yarom[1,6]

[1] University of Adelaide
fergus@beware.dropbear.id.au,yval@cs.adelaide.edu.au
[2] University of Pennsylvania
{gmicheli,danielg3,nadiah}@cis.upenn.edu
[3] University of Lübeck
thomas.eisenbarth@uni-luebeck.de
[4] Worcester Polytechnic Institute
amoghimi@wpi.edu
[5] University of Maryland
[6] Data61

Abstract. Intel Software Guard Extensions (SGX) allows users to perform secure computation on platforms that run untrusted software. To validate that the computation is correctly initialized and that it executes on trusted hardware, SGX supports attestation providers that can vouch for the user's computation. Communication with these attestation providers is based on the Extended Privacy ID (EPID) protocol, which not only validates the computation but is also designed to maintain the user's privacy. In particular, EPID is designed to ensure that the attestation provider is unable to identify the host on which the computation executes.

In this work we investigate the security of the Intel implementation of the EPID protocol. We identify an implementation weakness that leaks information via a cache side channel. We show that a malicious attestation provider can use the leaked information to break the unlinkability guarantees of EPID.

We analyze the leaked information using a lattice-based approach for solving the hidden number problem, which we adapt to the zero-knowledge proof in the EPID scheme, extending prior attacks on signature schemes.

Keywords: SGX, Side-Channel attacks, EPID, hidden number problem, Zero-Knowledge Proofs

1 Introduction

Mainstream processors have recently employed Trusted Execution Environments that allow running sensitive computation on potentially-compromised computers owned by untrusted third parties. The most prominent example is Intel Software Guard Extensions (SGX), which is a set of extensions to the Intel x86 architecture that aims to reduce the level of trust required of the platform owner at runtime. In particular, SGX is designed to enable secure computation under the assumption that the whole software stack, including the operating system (OS), is malicious. In order to achieve such strong security guarantees, SGX introduces runtime environments known as *enclaves* that are isolated from the software running on the computer. To ensure isolation, SGX strictly controls the entry to and exit from enclaves and limits inadvertent state transfer from enclaves to the outer untrusted software.

In order to allow the user to distinguish between legitimate and properly configured hardware and potentially corrupted software emulators, SGX supports a *remote attestation*

protocol that allows users to verify the legitimacy of the enclave before sending sensitive data to the enclave. The remote attestation protocol can verify if the enclave is running on a supported, and hence presumably trusted, processor and that the enclave has been initialized correctly. Remote attestation has been implemented through a combination of two *architectural enclaves*, the *provisioning enclave* and the *quoting enclave*, which together implement the Enhanced Privacy ID (EPID) protocol of [BL11]. EPID generates signatures that can be verified by a trusted *attestation server*. While the attestation server is trusted to verify the signatures, it is not trusted to maintain users' privacy. Consequently, SGX uses blinded group signatures which are unlinkable, allowing the attestation server to verify the signature's validity without knowing the signer's identity.

Microarchitectural side-channel attacks extract otherwise-unavailable secret information by artificially creating observable contentions between different CPU execution units. Since their introduction over a decade ago [Pag02, TTMH02, TSS$^+$03, Per05, Ber05, OST06], microarchitectural side channel attacks have been used to break the security of numerous implementations, including attacks on cryptographic primitives [ASK07, AS08, BvdPSY14], measurement of keystroke timings [LGS$^+$16, LGS$^+$17], website fingerprinting [GZES17], attacks from within the target's browser [OKSK15, AKM$^+$15, GMM16], from inside or against SGX enclaves [XCP15, LSG$^+$17, VBWK$^+$17, SWG$^+$17, MIE17, BMD$^+$17, MES17], or even on third party compute clouds [LYG$^+$15, IAES15, IGI$^+$16]. More recently, microarchitectural cache attacks have been combined with speculative execution in order to read sensitive data across security domains, such as kernel data [LSG$^+$18, KGG$^+$18].

Much less is known, however, about the effects of side channel attacks on long term privacy of SGX enclaves. In particular, while microarchitectural side channel attacks can be used to extract information from third-party software running within SGX enclaves [XLCZ17], the effects of side channel attacks on the enclave's own attestation mechanism have not been properly understood. Thus, in this paper we investigate the following questions:

How do side-channel attacks affect SGX's EPID attestation protocol? More specifically, can side-channel attacks be used to violate EPID's forward or backward privacy?

1.1   Our Contribution

A Side-Channel Attack on Intel's EPID Protocol.   In this work, we answer the above question in the affirmative, by presenting the first cache attacks on Intel's EPID protocol, as implemented inside SGX's quoting enclave. Our attack is able to recover part of the enclave's long term secret key, thereby allowing a malicious attestation server operator to break the unlinkability guarantees of SGX's remote attestation protocol.

Lattice Attacks on Zero Knowledge Proofs.   As we show in Section 3, Intel's implementation of the EPID protocol partially leaks the length of the randomness used for one of the zero-knowledge proofs used during EPID attestation. In order to recover part of the target's long term secret key, we use this information to build an instance of the hidden number problem (HNP) [BV96], which we solve using lattices. While such an approach was previously used [NS02, NS03, BT11, vdPSY15, GPP$^+$16] for the case of nonce leakage from digital signatures, in this paper we extend this approach from digital signatures to zero-knowledge protocols. To the best of our knowledge, this is the first application of side channels and lattice techniques beyond signatures, to the more general case of zero-knowledge protocols.

Attack Evaluation and Error Handling.   We evaluate the efficiency of our lattice attack in this setting, including measuring the effects of different optimizations, the tradeoffs involved in incorporating samples of different sizes into key recovery, and the robustness of the lattice construction against the types of measurement errors we encountered in our attack. In particular, we give experimental evidence that the lattice attack can still succeed even when a small number of erroneous traces are included, for the type of error

we observed in our measurements, where the side channel observation undercounted the true nonce length by several bits.

1.2 Targeted Software and Hardware

Attack Scenario. In line with previous attacks on SGX [LSG+17, MIE17], we assume that the attacker has root access to a Linux machine. The attacker can control the OS resources, including assignment of processes to cores, interleaved execution of SGX enclave and a cache monitoring process, as well as configuring the processor power and frequency scaling. While powerful, these are valid assumptions for attacking SGX enclaves as SGX excludes the OS from its trusted computing base and assumes that the OS is malicious.

As a motivating example for the attack scenario, we look at Signal's Private Contact Discovery Service[1]. This service allows clients to probe their contact list without revealing the probe results to any of the service operators. To protect the service, Signal implements it in an SGX enclave, and employs secret, unlinkable provisioning which hides the identity of the servers providing the service from the attestation provider. However, a malicious attestation provider can use our side-channel attack in conjunction with the information it gets as part of the EPID protocol to find the host's private key. This private key can then be used to link all attestation requests for services running on the host, exposing the service to the attestation provider.

To the best of our knowledge, at the moment Intel is the only attestation provider and thus our attack currently only allows Intel to break EPID's unlinkability property. However, in principle, SGX's design also allows for (less trusted) third-party attestation providers. Unless mitigated, our attack would apply to these parties as well.

Hardware. In principle, our attack is applicable to any CPU that supports Intel's SGX and EPID attestation. We empirically demonstrate our attack on a Dell Inspiron 5559 laptop with an Intel Skylake i7-6500U CPU featuring two hyper-threaded physical cores. Each physical core has 32 kB of L1D cache, used as our side channel. The L1 cache is 8-way associative and consists of 64 sets.

Software. Our target laptop is running Ubuntu 16.04 and SGX Software Development Kit (SDK) version 1.7. The targeted quoting enclave and EPID library matches between the Intel prebuilt binaries and the SGX SDK source code[2].

Because the EPID signatures in the quoting enclave implementation are encrypted to a hard-coded RSA public key before being transmitted to the remote attestation provider, as described in Section 3, we modified the quoting enclave to encrypt to our own public key so that we could decrypt the messages. This extra encryption in the implementation is not part of the EPID attestation protocol. Our threat model for this scenario is that the remote attestation provider is the attacker.

We implemented the side-channel attack using CacheZoom[3] [MIE17]. Our signal analysis heuristics are developed using Matlab version R2017a and its signal analysis toolbox.

Current Status. We notified Intel of our results in January 2018, and the vulnerability has been assigned CVE-2018-3691.

2 Preliminaries

2.1 Prime+Probe

Cache attacks exploit contention on a shared cache to infer secret information from unsuspecting processes. They can be harmful if an adversary and benign threads share access to the same cache. By measuring access time to shared or its own data, a spy process

---

[1] https://github.com/whispersystems/ContactDiscoveryService/
[2] SGX SDK is accessible at https://github.com/01org/linux-sgx
[3] CacheZoom source code is accessible at https://github.com/vernamlab/CacheZoom

can infer if a victim process has accessed some specific data or a related region of the cache. The resulting side channel leaks information about the memory access patterns of the victims which can be exploited to attack implementations of cryptographic schemes such as AES [OST06], RSA [AS08], and (EC)DSA [BvdPSY14]. Prime+Probe is a cache attack methodology that does not rely on any shared memory addresses between the attacker and the victim [Per05, OST06]. Prime+Probe has been demonstrated to be effective at stealing sensitive information, e.g. cryptographic keys, across virtual machines in cloud environments [LYG+15, IAES15]. Similarly, the Prime+Probe methodology scales well to the SGX threat model where an adversary, e.g. a malicious OS, shares the same cache while being unable to access enclave memory pages. The Prime+Probe methodology works as follows:

1. **Prime.** The attacker fills relevant cache sets by sequentially loading memory addresses that map to the same set.

2. **Victim Memory Access.** The attacker waits for the victim to perform secret-dependent memory operations. Memory operations by the victim evict some of the Attacker's cache lines from the targeted sets.

3. **Probe.** The attacker reloads the previously cached memory addresses and measures the access times to each cache set. A longer access time to a set corresponds to a victim access to that particular set. When the value of a secret variable affects the access patterns to memory, these pattern reveal information about the secret.

The cache hierarchy of modern Intel processors consists of three levels, with each level being larger and slower than the levels above it. The L1 cache at the top of the hierarchy is split into two caches: the L1 Data (L1D) cache is used for the data the program accesses; the L1 Instruction (L1I) cache stores the instructions that the programs execute. The L1D cache is virtually indexed, i.e. the processor uses the virtual address to find the cache set that stores the data. Consequently, by targeting the L1D cache in our attack, we avoid the need to map the cache as was required for past works that target the next level and larger caches [LYG+15, IGI+16, YGL+15].

2.2   Intel SGX

Intel Software Guard Extensions (SGX) are extensions of the Intel instruction set that provide trusted execution environments (TEEs) in Intel processors. The extensions, available since the Skylake processor generation [AMG+15], introduce secure execution environments called *enclaves*, that only include the processor hardware in their trusted computing base (TCB). Enclaves are protected through a combination of hardware measures that encrypt the enclave memory and strictly control the processor state at entry to and exit from the enclave.

Because SGX excludes the OS from its trusted computing base, the OS is assumed to be malicious. This, combined with SGX's lack of protection against side-channel attacks [AMG+15, CD16], have paved the way for stronger attack models that include adversarial control of the OS. Several side channels exploiting these adversarial powers have been demonstrated, including attacks on the page tables [XCP15, VBWK+17], branch target buffers (BTB) [LSG+17], caches [SWG+17, MIE17, BMD+17] and memory false dependency [MES17].

One notable technique that has been used across multiple attacks [LSG+17, MIE17] is exploiting the operating system's power to interrupt the enclave frequently. The technique allows the adversary to get information at a high temporal resolution and monitor memory accesses of almost every single instruction performed by the victim enclave.

Mechanisms to protect SGX enclaves against side channel attacks have also been proposed: page table attacks can be mitigated through compiler-level page table masking [SCNS16] or through minor modifications to the mechanism of page table entries (PTE) within the SGX hardware [SP17]. Other compiler-level protections have been proposed based on software diversity and binary code retrofitting to mitigate cache attacks [WWB+17, BCD+17]. *Déjà Vu* aims to detect excessive interruption introduced by OS adversaries [CZRZ17]. However, none of these mitigations have been adopted by the Intel provisioning enclave and quoting enclave. Further, the soundness and efficiency of these ad-hoc solutions have not been verified in practice.

Using an untrusted OS implies that users of the enclave need some mechanism to ensure that they communicate with a legitimate enclave that has been initialized correctly. To achieve this, Intel provides support for *remote attestation*. Remote attestation relies on a combination of secret keys stored within the processors and a cryptographic protocol which allows users to verify that they communicate with an enclave (as opposed to communicating with fake software set up by a malicious OS), that the enclave is running on a supported, and hence presumed trusted, processor and that the enclave has been initialized correctly. The protocol used for remote attestation, the Enhanced Privacy ID (EPID), is described in Section 2.4.

2.3 Bilinear Maps

Following the notation of [BBS04], let G_1 and G_2 be prime order multiplicative cyclic groups with generators g_1 and g_2 (respectively). We say that a mapping $e : G_1 \times G_2 \to G_T$ is an admissible bilinear map if the following properties hold.

1. **Bilinearity.** For all $(u, v) \in G_1 \times G_2$ and for all $a, b \in \mathbb{Z}$, it holds that $e(u^a, v^b) = e(u, v)^{ab}$.

2. **Non-Degeneracy.** $e(g_1, g_2)$ is a generator of G_T and $e(g_1, g_2) \neq 1$.

3. **Efficiently Computable.** It is possible to efficiently compute $e(u, v)$ for all $(u, v) \in G_1 \times G_2$.

As we focus on the EPID construction of [BL11], we now review two parameter choices for G_1, G_2, G_T. Indeed, in order to achieve 80-bit security level, [BL11] initialize G_1, G_2, G_T with a family of 170-bit non-supersingular elliptic curves defined by Miyaji et al. [MN01]. Next, for 128-bit security, [BL11] follows the suggestion of Koblitz and Menezes [KM05] and uses a 256-bit elliptic curve which has a suitable admissible bilinear map. As only the version offering 128-bit security is implemented in Intel's SGX SDK, we focus on this version of EPID which operates over the Fp256BN curve as standardized in [Int09]. However, our techniques are also applicable to the 80-bit version of EPID.

2.4 Enhanced Privacy ID

2.4.1 Overview

Enhanced Privacy ID (EPID) is a protocol proposed to allow remote attestation of a hardware platform without compromising the device's privacy [BL11]. EPID allows a platform to sign objects without exposing the platform identity to the verifiers, and it protects against adversaries who try to link multiple signatures to the same platform. Following the notation of [BL11], an EPID scheme consists of four entities. An Issuer $\mathcal{I}$, a revocation manager $\mathcal{R}$, a platform $\mathcal{P}$ and a verifier $\mathcal{V}$. The revocation manager maintains two revocations lists Priv-RL and Sig-RL. The scheme operates as follows:

Setup. First, the issuer $\mathcal{I}$ runs the scheme's setup algorithm Setup, on input 1^k, where k is the security parameter, obtaining a group public key gpk and a issuer secret key isk.

That is,

$$(\mathsf{gpk}, \mathsf{isk}) \leftarrow \mathsf{Setup}(1^k).$$

Join. Next, each platform $\mathcal{P}_i$ and the issuer $\mathcal{I}$ perform a Join protocol where $\mathcal{I}$'s input is $(\mathsf{gpk}, \mathsf{isk})$ and $\mathcal{P}_i$'s input is gpk. The Join protocol terminates with $\mathcal{P}_i$ learning a secret key sk_i which it will use to sign messages to the verifier $\mathcal{V}$. More formally,

$$\langle \bot, \mathsf{sk_i} \rangle \leftarrow \mathsf{Join}_{\mathcal{I},\mathcal{P}_i} \langle (\mathsf{gpk}, \mathsf{isk}), \mathsf{gpk} \rangle.$$

Sign. In order to sign a message m, using a group public key gpk, a secret key $\mathsf{sk_i}$, and a signature based revocation list $\mathsf{Sig\text{-}RL}$, the ith platform $\mathcal{P}_i$ runs the signing algorithm Sign. If $\mathsf{sk_i}$ is not revoked (i.e., $\mathsf{sk_i} \notin \mathsf{Sig\text{-}RL}$), Sign outputs a signature σ. Otherwise, Sign outputs $\bot$. Formally,

$$\bot/\sigma \leftarrow \mathsf{Sign}(\mathsf{gpk}, \mathsf{sk}_i, m, \mathsf{Sig\text{-}RL}).$$

Verify. To verify a signature σ on a message m using a group public key gpk, a private key-based revocation list $\mathsf{Priv\text{-}RL}$ and a signature-based revocation list $\mathsf{Sig\text{-}RL}$, the verifier $\mathcal{V}$ executes the verification algorithm Verify. This algorithm outputs valid or invalid. The invalid output indicates that either σ is not a valid signature on the message m or that the platform P_i signing m has been revoked. Formally,

$$\mathsf{valid/invalid} \leftarrow \mathsf{Verify}(\mathsf{gpk}, m, \mathsf{Priv\text{-}RL}, \mathsf{Sig\text{-}RL}, \sigma).$$

Revoke. EPID supports two types of revocations. In case the revocation manager $\mathcal{R}$ knows the private key of the platform $\mathcal{P}_i$ it wishes to revoke, it simply adds sk_i to the private key-based revocation list $\mathsf{Priv\text{-}RL}$ by running the revoke algorithm. This results in an updated private key-based revocation list. Formally, $\mathcal{R}$ performs

$$\mathsf{Priv\text{-}RL} \leftarrow \mathsf{Revoke}(\mathsf{gpk}, \mathsf{Priv\text{-}RL}, \mathsf{sk}_i).$$

Moreover, if $\mathcal{R}$ wants to revoke some platform based on a message-signature pair (m, σ) it signed but without knowing its secret key, he can also execute the Revoke algorithm using $\mathsf{gpk}, \mathsf{Priv\text{-}RL}, \mathsf{Sig\text{-}RL}, m, \sigma$ as inputs. This results in an updated signature-based revocation list $\mathsf{Sig\text{-}RL}$. Formally,

$$\mathsf{Sig\text{-}RL} \leftarrow \mathsf{Revoke}(\mathsf{gpk}, \mathsf{Priv\text{-}RL}, \mathsf{Sig\text{-}RL}, m, \sigma).$$

2.4.2   Security Properties of EPID

In this section we briefly overview the security properties of the EPID scheme of [BL11]. We refer the reader [BL11] for formal discussions and definitions.

Correctness. Informally, the correctness requirement states that every signature σ generated by a platform $\mathcal{P}_i$ on a message m with a secret key sk_i is valid, unless $\mathcal{P}_i$ has been revoked. More formally, let Σ_i be the set of all signatures generated by $\mathcal{P}_i$. We require that $\mathsf{Verify}(\mathsf{gpk}, m, \mathsf{Priv\text{-}RL}, \mathsf{Sig\text{-}RL}, \mathsf{Sign}(\mathsf{gpk}, \mathsf{sk}_i, m, \mathsf{Sig\text{-}RL})) = \mathsf{valid}$ if and only if $\mathsf{sk}_i \notin \mathsf{Priv\text{-}RL}$ and $\Sigma_i \cap \mathsf{Sig\text{-}RL} = \emptyset$.

Unlinkability. At a high level, EPID's unlinkability requirement guarantees that it is impossible to identify the platform that produced a signature σ on some message m, nor is it possible to identify other signatures signed by the same platform. More specifically, even in case a malicious issuer colludes with a malicious verifier (such as in the case of a malicious remote attestation provider), knowing $\mathsf{gpk}, \mathsf{isk}$ and a list of message signature pairs $(m_i, \sigma_i)_{i=1,\cdots,n}$ is not sufficient for linking a pair m, σ to a specific secret key sk or to other signatures signed by the same secret key.

Unforgeability. At a high level, EPID's unforgeability requirement states that it is impossible for the attacker to forge a valid signature on some previously-unsigned message, without knowing a non-revoked secret key. This holds even in the case when the attacker knows previously revoked secret keys, belonging to compromised platforms. We refer the reader to [BL11] for a more formal discussion.

2.4.3 The Signing Algorithm

In this work, we attack the signing algorithm used by $\mathcal{P}$ to sign a message m. We will give a high-level description of EPID's signing algorithm in order to motivate the attack. We refer the reader to [BL11] for the full details.

Let p be the order of the bilinear group pair (G_1, G_2) and let $e : G_1 \times G_2 \to G_T$ be an admissible bilinear pairing (as defined in Section 2.3). The secret key sk_i used by the ith platform $\mathcal{P}_i$ to sign m consists of $\mathsf{sk}_i = (A, x, y, f)$, where f is a random element of $\mathbb{Z}_p$ and (A, x, y) is a BBS+ signature [ASM06] on f. Following [BL11] and [CS97], we use the notation $SPK\{(a) : y = z^a\}(m)$ to denote a signature on a message m where the signature scheme used to sign m is derived from some interactive zero knowledge proof-of-knowledge protocol via the Fiat-Shamir heuristic. In this notation, the parenthesized values (a) are known to the platform $\mathcal{P}_i$ but not to the verifier $\mathcal{V}$ while all other values are public and thus known to both $\mathcal{P}_i$ and $\mathcal{V}$. Let $\mathsf{Sig\text{-}RL}$ be a signature-based revocation list, at a high level the algorithm proceeds as follows:

1. Choose randomly $B \leftarrow G_3$ and compute $K := B^f$, where G_3 is a cyclic group of order p defined as part of the scheme group public key gpk.

2. Choose a random $a \leftarrow \mathbb{Z}_p$, and compute $b \leftarrow y + ax$, and $T \leftarrow A \cdot h_2^a$ where $h_2 \in G_2$ is also part of the scheme group public key gpk.

3. Run the following signature of knowledge protocol

$$SPK\{(x, f, a, b) : B^f = K \wedge e(T, g_2)^{-x} \cdot e(h_1, g_2)^{r_f} \cdot e(h_2, w)^{r_a} = e(T, w)/e(g_1, g_2)\}(m)$$

   where

   (a) $r_x \leftarrow \mathbb{Z}_p, r_f \leftarrow \mathbb{Z}_p, r_a \leftarrow \mathbb{Z}_p$, and $r_b \leftarrow \mathbb{Z}_p$ are chosen uniformly at random.

   (b) $c \leftarrow H(\mathsf{gpk}, B, K, T, R_1, R_2, m)$, where H is a cryptographic hash function.

   (c) $s_x \leftarrow r_x + cs$, $s_f \leftarrow r_f + cf$, $s_a \leftarrow r_a + ca$, and $s_b \leftarrow r_b + cb$ where all computations are performed over $\mathbb{Z}_p$.

   (d) The values g_1, g_2, h_1, h_2 are specified in the group public key gpk.

4. Set $\sigma_0 = (B, K, T, c, s_x, s_f, s_a, s_b)$.

5. Let $\mathsf{Sig\text{-}RL} = \{(B_1, K_1), \ldots, (B_{n_2}, K_{n_2})\}$. For all i, compute $\sigma_i = SPK\{(f) : K = B^f \wedge K_i \neq B_i^f\}(m)$.

6. If any zero-knowledge proofs in Step 4 fails, then output $\sigma = \perp$. Otherwise, output the signature $\sigma = (\sigma_0, \sigma_1, \cdots, \sigma_{n_2})$.

Notice that if the attacker is able to leak the value of f by mounting a side-channel attack on the exponentiation routine, he is able to break the unlinkability property of the EPID construction by linking $\mathcal{P}$ to all its signatures, including past and future signatures. Next, notice that in Step 3 of the above description, the platform $\mathcal{P}$ generates a random secret nonce $r_f \in \mathbb{Z}_p$ and computes an exponentiation $e(h_1, g_2)^{r_f}$ (where $h_1 \in G_1$ and $g_2 \in G_2$ were both published in gpk). The signature component σ_0 includes the value $s_f = r_f + cf$ where c is the hash over several public values generated during the signing process.

As we show in Section 5, by recovering additional side-channel information about the length of the nonce r_f from several such signature operations, we are able to mount a lattice attack on the EPID construction and completely recover the value of f. Since the values of B and K are also part of σ, we are able to break the unlinkability property of the EPID construction by checking weather B^f equals K.

3   SGX EPID Provisioning and Attestation

Intel implements the EPID attestation through a combination of two enclaves, collectively known as the architectural enclaves, and deployed as part of the SGX SDK. The *provisioning enclave* is responsible for performing the EPID Join operation, effectively verifying the SGX hardware to the Intel key facility and obtaining the attestation key. The *quoting enclave* implements the EPID Sign operation. It uses the previously obtained attestation key to generate the signature attestation on each request. Note that the provision operation only needs to be performed once when a trusted environment is being prepared.

3.1   Provisioning and Quoting Enclave Implementations

The provisioning enclave is signed by Intel and has a special attribute that allows it to derive the permanent provision key from the hardware fused provision secret. Intel SGX supports multiple keys for different operations that can be retrieved using a special CPU instruction. Two of these keys, the provision key and the provision seal key, are only accessible to the provisioning enclave. The provisioning enclave uses an ECDSA operation and the provision key to sign a message that authenticates the SGX hardware to the Intel provisioning servers. However, the signing operation does not directly use the 128-bit provision key. Instead, a key-wrapping operation using AES-CMAC [SPLI06] over a hardcoded plaintext generates the 256-bit ECDSA key. We skip the explanation of the entire authentication protocol between the provisioning enclave and Intel provisioning server. After the authentication, the provisioning enclave obtains the private attestation key f. The provisioning enclave uses the provision seal key to store f on the disk in an encrypted form.

The quoting enclave unseals the attestation key stored by the provisioning enclave on each attestation request. The attestation process follows the EPID scheme and results in an EPID signature. However, in the quoting enclave implementation, this signature is encrypted using a hybrid RSA-AES-CMAC based on a hardcoded RSA public key. While this operation is not part of the EPID scheme it does add a layer of security to the EPID signatures used in SGX.

From an attacker perspective, this encryption hides the otherwise public EPID signature. Consequently, to be able to mount the attack we describe below, the attacker needs to be able to decrypt the EPID signatures. To allow us to decrypt the signatures, we modify the quoting enclave to use our own public key, for which we know the private key. We then sign the modified enclave. In order to avoid potential changes due to differences in the build environment, the change is applied to the binary file, rather than to the source. Hence, aside from the RSA public key and the enclave metadata, our quoting enclave is identical to the original.

We note that to perform the EPID verification, the attestation provider must be able to decrypt the EPID signatures. Hence, a real attack is only possible from a malicious attestation provider. Nevertheless, one of the main aims of the EPID protocol is to protect the privacy of the users and prevent a malicious attestation provider from linking signatures to the hosts that signed them. Hence, our attack enables a malicious attestation provider to break one of the main objectives of using the EPID protocol.

3.2   Scalar Multiplication in the Quoting Enclave

To perform a scalar multiplication, the quoting enclave recodes the scalar s using an extension of the Booth recoding [Boo51] and uses a fixed-window algorithm with a window size of 5 and the recoded scalar. Recoding with a window size w represents the scalar as a sequence of digits s_i such that $-2^{w-1} \le s_i \le 2^{w-1}$ and $s = \sum_i 2^{wi} s_i$.

The algorithm (see Algorithm 1) precomputes the values $P_i = P^i$ for an input point P and $0 \le i \le 2^{w-1}$. It then scans the scalar from the most significant non-zero digit to the least significant digit. For each digit, it performs w squaring operations of an

---

Algorithm 1 Scalar Multiplication in the Quoting Enclave

1: **procedure** MULPOINT(point P, window size w, scalar s represented as $s_0 \ldots s_n$ using the Booth recoding)
2:     $P_0 \leftarrow \mathcal{O}$
3:     **for** $i \leftarrow 1$ **to** 2^{w-1} **do**
4:         $P_i \leftarrow P \cdot P_{i-1}$
5:     **end for**
6:     $i \leftarrow \max\{j : s_j \neq 0\}$
7:     $r \leftarrow P_{|s_i|}$
8:     $i \leftarrow i - 1$
9:     **while** $i \geq 0$ **do**
10:         $r \leftarrow r^{2^w}$
11:         $t \leftarrow P_{|s_i|}$
12:         $r \leftarrow r \cdot \text{ctSelect}(t, t^{-1}, \text{isNegative}(s_i))$
13:         $i \leftarrow i - 1$
14:     **end while**
15:     return r
16: **end procedure**

---

intermediate result r followed by a multiply of the precomputed value matching the value of the digit or its inverse (Line 12). To protect against cache attacks, the algorithm uses a constant-time select operation for the decision whether to use the precomputed value or its inverse. Furthermore, it uses the scatter-gather approach [GGO+09, BGS06] to mask the cache fingerprint of accesses to the precomputed values. For simplicity, we omit this scatter-gather operation from the algorithm.

Despite the countermeasures taken, the algorithm leaks the length of the recoded representation of the scalar. More specifically, Algorithm 1 starts from the most significant non-zero digits (Line 6), leaking the length of the Booth representation of the scalar. The length of the recoded representation corresponds to the number of leading zero bits in the scalar. A full-length recoded scalar has 52 digits and the main loop of Algorithm 1 iterates 51 times. When both bits 255 and 256 of the scalar are 0, the recoded scalar has 51 digits and the loop iterates 50 times. Each additional five clear bits correspond to shortening the recoded scalar by one digit and to a resulting decrease in the number of iterations through the loop. Thus an adversary who can count the number of iterations through the main loop of Algorithm 1 or accurately measure the time it takes to perform the scalar multiplication can recover the values of some of the most significant bits of the scalar.

4 Short Scalar Leakage via High Resolution Side Channels

In order to extract the key leakage of EPID from an SGX quoting enclave we monitor the number of loop iterations via the L1 data cache, which is a convenient channel providing high measurement resolution.

4.1 Controlled Prime+Probe Attack

In this attack, we follow the scenario of [MIE17] and apply a high-resolution Prime+Probe attack in a controlled environment with respect to the OS adversarial model. More specifically,

1. The processor is configured to operate on a constant frequency to avoid dynamic changes of frequency. This reduces noise by making time measurements more accurate.

2. The thread running the quoting enclave and the Prime+Probe code is isolated on one physical core, while all other tasks of the system are placed on other physical cores of
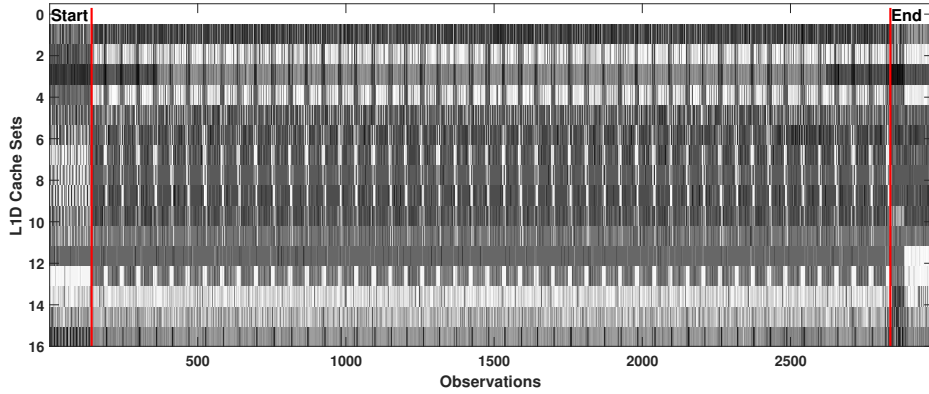
Figure 1: Heatmap of 16 different cache sets for the scalar multiplication. Each cache set that has a repetitive memory pattern, for example sets 7, 9 and 13, shows 48 repetitions, indicating that the ephemeral key bit size is less than $(48 + 1) \cdot 5 = 245$ bits. The red lines mark the start and the end of the repetitive memory pattern.

the system. This removes noise caused by memory activities of irrelevant operations from the monitored core and its L1 caches.

3. The timer interrupt handler on the attack core is configured to be triggered with a very high frequency. Thus, the quoting enclave thread can only execute a few instructions between each interrupt. In the interrupt handler, we perform Prime+Probe on the 64 L1D cache sets. As the target quoting enclave is only able to perform a few memory operations in each time frame between two consecutive interrupts, the Prime+Probe reveals all memory accesses of the enclave with high temporal and spatial resolution.

Figure 1 shows the observations of 16 different cache sets for the quoting enclave Booth multiplier. Each loop of the multiplier executes several memory operations affecting different cache sets at different times in a periodic way. As a result, we can count the number of iterations of the main loop of Algorithm 1 by looking at any cache set that has a periodic memory access pattern. Because the main loop performs one iteration for each w-bit digit following the first non-zero digit, counting the number of iteration provides information on the bit length of the scalar.
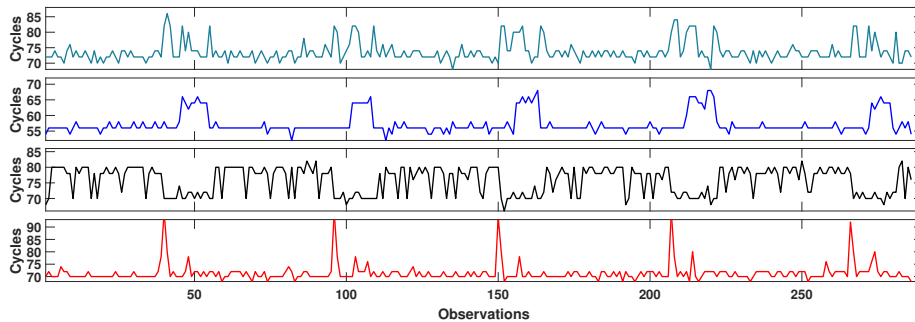

Figure 2: Cache access patterns on four different sets during the computation of the main loop of Algorithm 1. Each set features a different repeating pattern of the same length that can be used to count the number of loop iterations executed by the quoting enclave.
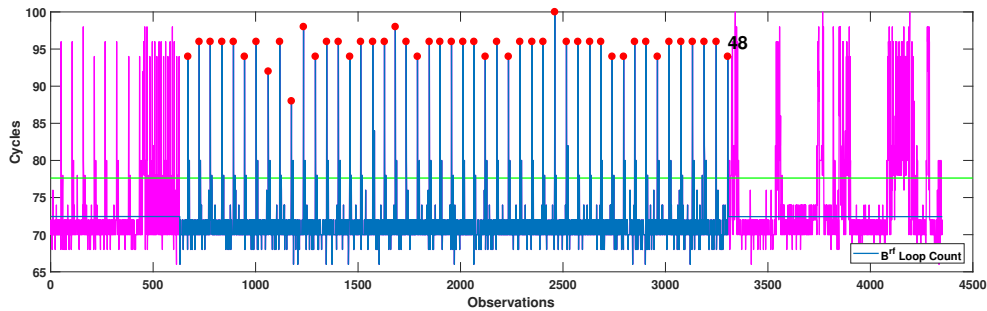
Figure 3: Counting loop iterations on set `02`: The number of equally spaced high peaks within a defined signal pattern reveals the number of loop iterations to be 48 in this example.

4.2 Loop Counting Analysis

Our goal is to determine the loop count for the Booth recoding of the scalar r_f using the above-described side-channel setup and to detect signatures that have been generated using short scalars. Our attack setup is configured to start the interrupted Prime+Probe measurement right before the call to the quoting enclave and to finish right after the enclave exits. The observations are stored in a circular buffer capable of recording 50,000 samples. The loop repetition leakage affects several cache sets in different ways, as shown in Figure 2.

Automatically Counting Loop Iterations. To automate the extraction of the number of loop iterations, we implemented several heuristics using the Matlab signal processing toolbox. A first layer locates the window for the start and end of the multiplier within the trace of 50,000 sample points. A second filter counts the number of repeated cache access patterns within the relevant window, which directly corresponds to the number of executed loops of the scalar multiplication. This information can be extracted from the periodic leakage in several of the cache sets. As depicted in Figure 2, the signal pattern from the main loop of Algorithm 1 is unique for each cache set. We use five different loop counters that use the information from four cache sets to count the number of loops on each signature. The first four counters detect periodic behavior to each of the four cache sets separately, while the last counter performs a reverse check on set `02`. Figure 3 shows a successful loop counting on set `02` that returns a count of 48 iterations, thus revealing the 12 most significant bits of the scalar are clear.

Handling Measurement Noise. Even though the L1D cache channel has a very high resolution, there is still some noise that can result in a failure to count the correct number of loop iterations for each of the five counters. Common sources of error are failing to accurately detect the beginning and the end of the multiplier window, under-counting short peaks and over-counting occasional noises that introduce unexpected peaks or pattern within the sampling of a multiplier window. However, our experimental results show that if four of the five loop counters agree on the number of loop iterations, the loop counting would be error free. We automatically analyzed 11,080 signature operations and found 1214 *50-loop*, 39 *49-loop* and 2 *48-loop* short keys without any error.

Further Reducing the Number of Signatures via Manual Processing. Although the fully automated approach returns enough short-key signatures to recover the key, it discards many samples that carry valuable information. Scalars resulting in a *49-loop* occur with a probability of 1/128. Thus, the automated approach, while ensuring no false positives, only detects about 45% of such occurrences. The number of required signature observations can be reduced by combining the automatic loop counting with manual verification. By returning all traces where three or more counters agree, the automatic

loop counting returns 1723 *50-loop*, 54 *49-loop* and 5 *48-loop* candidates, with 0.4%, 5.0% and 0.0% error, respectively. Further reducing to the minimum of two matching counters yields 2155 *50-loop*, 117 *49-loop* and 7 *48-loop* candidates, with 5.7%, 52% and 14% error, respectively. In this case, manual post-processing is necessary, but certainly tractable, e.g. for the 117 *49-loop* candidates. We manually verified 59 of the *49-loop* candidates, thus increasing the ratio of found occurrences of *49-loop* candidates to 68%.

5   A Lattice Attack on EPID

The side channel gives us information about the length of r_f used to compute the signature component $s_f = r_f + cf \bmod p$, where s_f and c are public information, p is the 256-bit order of the elliptic curve, and f is the platform's secret membership key. We will use this information to solve for f.

5.1   The Hidden Number Problem

In the hidden number problem (HNP) [BV96], there is a secret integer α, and the attacker is given many samples from the l most significant bits of random multiples of $\alpha \bmod p$. For a given prime p and a fixed l, Boneh and Venkatesan showed how to recover the secret α in polynomial time with probability greater than $1/2$, under the assumption that $|\alpha t_i - u_i| < p/2^l$, where t_i are uniformly and independently randomly chosen integers in $\mathbb{Z}_p^*$ and the u_i are integers representing the knowledge of the most significant bits of $\alpha t_i \bmod p$.

Boneh and Venkatesan originally formulated the hidden number problem to prove the existence of hardcore bits for the Diffie-Hellman key exchange [BV96]. Their work has since been extensively used for different attacks. Nguyen and Shparlinski used HNP to attack the DSA and ECDSA signing algorithms [NS02, NS03]. They use the Boneh-Venkatesan lattice construction. Since their work, a number of papers have demonstrated attacks on the OpenSSL implementation of the ECDSA digital signature algorithm [FWC16, ABF+16, BT11]. Researchers have also introduced new variants of the HNP, such as the modular inversion hidden number problem [BHHG01] and the extended hidden number problem that considers chunks of known bits [HR06]. The problem we consider fits into the original hidden number problem setting. We begin by explaining how to convert our problem to a hidden number problem instance.

5.2   Conversion to a Hidden Number Problem

In the following, we will drop the subscript f from the signature component, and use a subscript i to index the sample number. We obtain many samples $\{(s_i, c_i)\}_i$ satisfying

$$s_i \equiv r_i + c_i f \bmod p, \tag{1}$$

as well as information about whether the number of most significant zero bits in r_i is 0, 2, 7, or 12. That is, we learn that $|s_i - c_i f| < p/2^{l_i}$ where l_i is the number of 0 bits we learn from r_i; this is identical to the hidden number problem as described above.

Recentering the Nonces.   Naively, the number of bits of r_i that we learn is 2 bits for a *50-loop* sample, 7 bits for a *49-loop* sample, and 12 bits for a *48-loop* sample. However, these r_i values are positive, while the lattice construction works with both positive or negative values for r_i. Since we know the length of the r_i, $r_i \le 2^{n_i}$, where $n_i = 256 - l_i$ in our application, we can recenter the r_i around 0 to reduce the size of our solution by one bit. That is, we can rewrite Equation 1 as $s_i' - r_i' \equiv c_i f \bmod p$, with $s_i' = s_i - 2^{n_i}$ and $r_i' = r_i - 2^{n_i}$. In this way, we obtain a new problem with $-p/2^{l_i+1} \le r_i \le p/2^{l_i+1}$. (See [NS02, NS03] for more details.) The effect of recentering the nonces on the success probability of the attack can be seen in Figure 4.

5.3   Solving the Hidden Number Problem

There are two standard approaches to solving the hidden number problem: via lattices or via Fourier analysis [IEE00, MHMP13]. We use the lattice-based approach in this paper, since it is quite efficient in the case of relatively large numbers of bits known (2, 7, or 12 in our attacks) and relatively few samples.

5.3.1   Lattice Preliminaries

A lattice is a discrete additive subgroup of $\mathbb{R}^n$. The set of vectors in a lattice is closed under addition and multiplication by integers. The discreteness property ensures that there is some $\lambda > 0$ such that the length of the shortest vector v_1 in the lattice satisfies $|v_1|_2 = \lambda$. More generally, we will use λ_i to denote the ith successive minimum of the lattice. Any n-dimensional lattice can be specified by a *basis* of at most n linearly independent vectors $\{b_i\}_{i=1}^n$. A convenient representation for a lattice basis is in the form of a matrix B whose rows are the basis vectors b_i. A lattice basis is not unique; one lattice basis can be transformed to a different basis for the same lattice by means of integer row operations.

The BKZ algorithm [Sch87, SE94] runs in time exponential in a block size k and polynomial in the lattice dimension n, and gives a reduced lattice basis whose vectors b_i satisfy the approximation $|b_i|_2 \leq i\gamma_k^{(n-i)/(k-1)}\lambda_i$ [Sch92], where γ_k is the Hermite constant. In practice, Chen and Nguyen [CN11] observe that BKZ typically returns vectors satisfying $b_1 \leq (1 + \epsilon_k)^n \lambda_1$ where ϵ_k depends on the block size; for random lattices they obtain $1 + \epsilon_k = 1.01$ for a block size of 85.

5.3.2   Solving the Hidden Number Problem via CVP embedding

The lattice-based approach to the hidden number problem was introduced by Boneh and Venkatesan [BV96] and recovers the nonces r_i and the secret f by solving a closest vector problem (CVP) in a given lattice. The idea is that given Equation 1 and the bound $|r_i| < p/2^{l_i}$, a vector obtained from the s_i will be particularly close to a vector containing the secret f.

However, current lattice algorithm implementations for solving the shortest vector problem (SVP), and more generally, computing a reduced basis for a lattice, are much better than implementations of CVP algorithms, so most practical attacks using these techniques embed the Boneh-Venkatesan lattice basis into a basis one dimension larger so that the desired closest vector solution in the original lattice corresponds to a particularly short vector in the new, slightly larger lattice. For example, given m recentered samples, Benger et al. [BvdPSY14] hope to recover the secret key by computing a short vector in the $(m + 2)$-dimensional lattice generated by the following basis. The northwest quadrant is a scaled version of the basis given by Boneh and Venkatesan, and the left portion of the bottom row vector is the target for CVP.

$$
B_{cvp} = \begin{bmatrix}
2^{l_1+1}p & \cdots & & & 0 & 0 \\
0 & 2^{l_2+1}p & & & 0 & \vdots \\
& & \ddots & & & \\
& & & 2^{l_m+1}p & 0 & \\
2^{l_1+1}c_1 & 2^{l_2+1}c_2 & \cdots & 2^{l_m+1}c_m & 1 & 0 \\
2^{l_1+1}s_1 & 2^{l_2+1}s_2 & \cdots & 2^{l_m+1}s_m & 0 & p
\end{bmatrix}. \tag{2}
$$

Benger et al. note that the shortest vector in this lattice is actually $(0, \ldots, 0, p, 0)$ and hope that the second-shortest vector in a suitably reduced basis is the target $v_2 = (2^{l_1+1}r_1, \cdots, 2^{l_m+1}r_m, f, -p)$. ($v_2 = zB$ for some $z = (z_1, \ldots, z_m, f, -1)$ with $z_i \in \mathbb{Z}$.) We can use LLL or BKZ to compute a reduced basis. We have $\det L(B) = 2^{m+\sum_i l_i} \cdot p^{m+1}$ and $|v_2|_2 \approx \sqrt{m+2}p$, and we hope to find v_2 when $|v_2|_2 \leq (1+\epsilon)^{m+1}(\det B)^{1/(m+2)}$ where $1 + \epsilon$ is the approximation factor achieved by the lattice basis reduction algorithm we use.
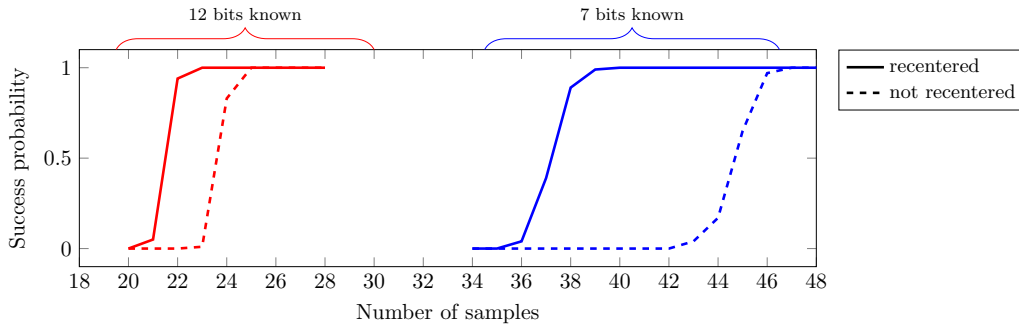
Figure 4: We show the success probability of key recovery using *49-loop* samples, which reveal the 7 most significant bits of the nonces, and *48-loop* samples, which reveal the 12 most significant bits learned, for different numbers of samples. Recentering the nonces has a noticeable impact on the number of samples required for key recovery.

Figure 4 shows the experimental success probability of the attack for *49-loop* samples, where 7 most significant bits of the nonce are known, and *48-loop* samples, where 12 most significant bits of the nonce are known.

5.4   Performance Tradeoffs

5.4.1   Using Samples of Different Lengths

Benger et al. [BvdPSY14] describe how to take advantage of different length samples using the CVP embedding construction by using different values for each l_i in the lattice basis given in Equation 2. Similarly, for our attack, this allows us to reduce the total number of samples we need to collect by using samples with different loop lengths. This results in a performance tradeoff: we can decrease the signature sampling time at the cost of increasing the time spent running lattice basis reduction to recover the secret key.

Using only *49-loop* samples in the lattice, corresponding to learning 7 most significant bits of the nonce, we needed 38 samples to achieve above a 50% success rate in the lattice construction; when we added 30 *50-loop* samples (in which we learn 2 most significant bits) to the lattice, we had above a 50% success rate with 30 *49-loop* samples. We show the improvement as the number of samples increases in Figure 5. Each point represents 100 trials using BKZ with block size 30; for the 73-dimensional lattices containing 31 *49-loop*
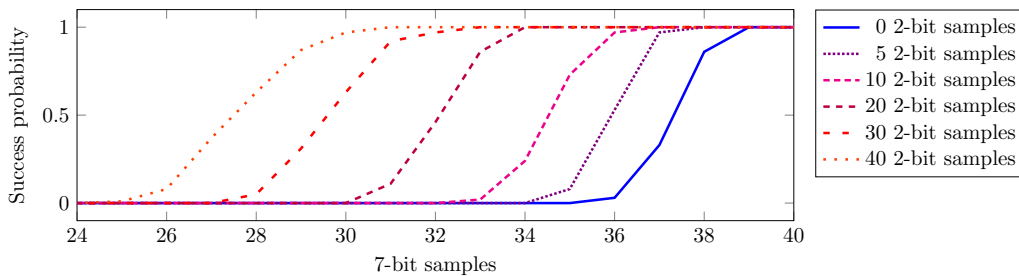

Figure 5: We can reduce the total number of signatures needed to carry out a successful attack by including samples of different sized nonce observations in the lattice. We constructed different sized lattices by including both samples that had revealed 7 bits of the nonce and samples that had revealed 2 bits of the nonce. The lattice dimension for each experiment is the total number of samples + 2. We measured the success probability over 100 trials on random problem instances, solved using BKZ-30.
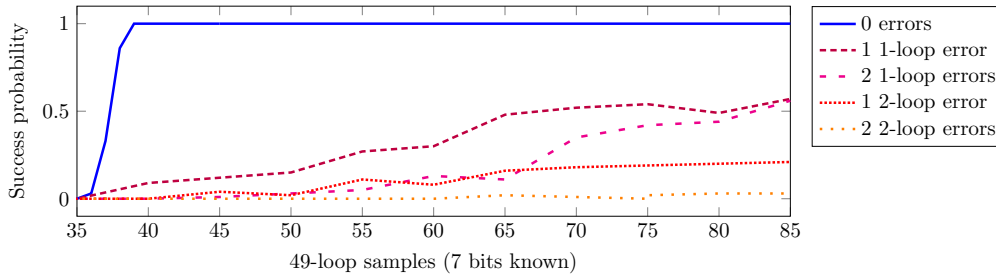
Figure 6: The lattice still finds the correct solution some fraction of the time, even in the presence of collection errors for the error model in our attack. We show the probability of successful recovery when the lattice contains a number of randomly generated samples whose actual nonce sizes are one or two windows, or 5 bits or 7 bits larger than the bound given in the lattice. Each point represents 100 randomly generated trials of random problem instances, solved using BKZ-30.

samples and 40 *50-loop* samples (corresponding to a 100% success rate) the BKZ algorithm took 52 seconds on average to complete on a single core of our test machine.

5.4.2 Error Correction

It is quite common in a side-channel attack that there is some error during the collection process. In the case of our attack, an error while counting the number of loops during the modular exponentiation would result in an incorrect bound on the size of one or more of the r_is being collected. If the loop count is higher than the actual value, this is not a problem: either the sample would be excluded from the key recovery process, or the size of the r_i is still correct for the bound we use in the lattice and we expect to still find the correct solution. However, if the error happens in the other direction and the measurement process undercounts the number of loops, we will incorporate samples into the lattice whose nonce sizes are larger than the bounds we assign for them. In this case, the lattice construction can fail, because the vector may no longer correspond to the correct key.

Much of the prior work on side-channel attacks with errors for similar constructions either ignored this issue entirely, dealt with it via signal processing, or subsampled different subsets of samples until an error-free sample is obtained [GPP+16, ARAM17].

However, we find experimentally that when the lattice includes more samples than necessary, key recovery may still be possible in the presence of the types of errors we encountered in our attack. In our measurements, an error corresponds to an incorrect loop count. To model this in experiments, we generated instances of *49-loop* samples, and inserted errors corresponding to samples that should have been measured as *50-loop* or *51-loop* samples. Recall that the nonce in a *49-loop* sample is contained in the interval $[2^{244}, 2^{249})$; we generated *50-loop* errors uniformly from the interval $[2^{249}, 2^{254})$ and *51-loop* errors uniformly from the interval $[2^{254}, p)$, inserted these into our samples, and attempted key recovery. Figure 6 shows the success probability of our lattice construction when the lattice contains different numbers of errors. Each plotted sample represents 100 randomly generated trials with the specified type of error run with BKZ-30. The recovery rate with errors whose most significant bits are 1 is similar to the error rates for the *51-loop* samples.

As a concrete example, a 75-sample lattice with 2 1-loop errors succeeded 42% of the time in our experiments. This corresponds to a 2.9% error rate. We would expect a 39-sample lattice (which succeeded with 100% probability in our experiments) to achieve 0% errors with probability $0.97^{39} = 0.35$, or 35% of experiments.

5.5   Recovering f

Using the automatically classified data from the side-channel attack described in Section 4, we were able to recover f using BKZ-30 on 37 error-free *49-loop* samples, corresponding to 7 most significant bits of each ephemeral r_i known. It took 10,600 total signature samples to collect this data. Next, by using samples of different loop lengths we can further reduce the total number of signatures required. More specifically, in Table 1 we show some different strategies for successful key recovery given our empirical signature data. For the row corresponding to n signature data points, we used all of the *48-loop* (corresponding to 12 bits known) and *49-loop* (corresponding to 7 bits known) samples that were detected during the first n signature measurements in our data, and then added *50-loop* (corresponding to 2 bits known) samples from the first n signatures to the lattice until key recovery succeeded.

As decribed in Section 4.2, fewer signatures are required if manual inspection is used to help classify the signals. In that case, we would only need less than 7,500 observed signatures to obtain enough *49-loop* observations for a full key recovery.

Table 1: Strategies for key recovery with different numbers of signature samples.

Signatures	48-loop	49-loop	50-loop	BKZ block size	BKZ time
10300	2	35	0	2	0.1s
10000	2	31	10	20	0.2s
9000	2	29	21	30	1.4s
8000	2	25	35	30	4.5s

6   Conclusions

In this work, we show yet another leakage in highly sensitive code—the implementation of the EPID protocol for SGX remote attestation. While the attack only allows a malicious attestation provider to break the link signatures to a host, the unlinkability guarantee it breaks is the main reason for using the EPID protocol in the first place. From the structure of the code, it is clear that the developers have attempted to eliminate side channel leaks. Thus, this incident demonstrates that producing constant-time code is not trivial and that better tools for facilitating such development are required.

This work extends the art of recovering the long-term key from partial information on the ephemeral keys. First, we apply known techniques used in the context of digital signatures to the wider context of zero-knowledge proofs. Second, we investigate the handling of erroneous inputs for the hidden number problem. We show that prior common belief notwithstanding, lattices can handle some erroneous input. Exploring the trade-offs between past approaches of selecting a random subset of the inputs and the new approach of using the inputs in the lattice is left for future work.

Acknowledgments

References

[ABF+16] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, pages 422–435, 2016.

[AKM+15] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE Symposium on Security and Privacy*, pages 623–639, 2015.

[AMG+15] Ittai Anati, Frank McKeen, Shay Gueron, Haitao Huang, Simon Johnson, Rebekah Leslie-Hurd, Harish Patil, Carlos Rozas, and Hisham Shafi. Intel software guard extensions (Intel SGX). Tutorial Slides presented at ISCA, June 2015.

[ARAM17] Jiji Angel, R. Rahul, C. Ashokkumar, and Bernard Menezes. DSA signing key recovery with noisy side channels and variable error rates. In *INDOCRYPT*, volume 10698 of *Lecture Notes in Computer Science*, pages 147–165, 2017.

[AS08] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *CT-RSA*, pages 256–273, 2008.

[ASK07] Onur Acıiçmez, Werner Schindler, and Çetin Kaya Koç. Cache based remote timing attack on the AES. In *CT-RSA*, pages 271–286, 2007.

[ASM06] Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA. In *SCN*, pages 111–125, 2006.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, pages 41–55. Springer-Verlag, 2004.

[BCD+17] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR. SGX: Hardening SGX enclaves against cache attacks with data location randomization. *arXiv preprint arXiv:1709.09917*, 2017.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. Preprint available at http://cr.yp.to/papers.html#cachetiming.

[BGS06] Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. Mitigating cache/timing based side-channels in AES and RSA software implementations. RSA Conference 2006 session DEV-203, February 2006.

[BHHG01] Dan Boneh, Shai Halevi, and Nick Howgrave-Graham. The modular inversion hidden number problem. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 36–51. Springer, 2001.

[BL11] Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *IJIPSI*, 1(1):3–33, 2011.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[Boo51] Andrew D. Booth. A signed binary mutiplication technique. *Q. J. Mech. Appl. Math.*, 4(2):236–240, January 1951.

[BT11]   Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *ESORICS*, pages 355–371, 2011.

[BV96]   Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *CRYPTO*, pages 129–142, Santa Barbara, CA, US, August 1996.

[BvdPSY14]   Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh aah... just a little bit": A small amount of side channel can go a long way. In *CHES*, pages 75–92. Springer, 2014.

[CD16]   Victor Costan and Srinivas Devadas. Intel SGX explained. IACR Cryptology ePrint Archive, Report 2016/086, 2016.

[CN11]   Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.

[CS97]   Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups (extended abstract). In *CRYPTO*, pages 410–424, 1997.

[CZRZ17]   Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *AsiaCCS*, pages 7–18. ACM, 2017.

[FWC16]   Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL implementation of ECDSA with a few signatures. In *CCS*, pages 1505–1515. ACM, 2016.

[GGO+09]   Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. Fast and constant-time implementation of modular exponentiation. In *Embedded Systems and Communications Security*, September 2009.

[GMM16]   Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *DIMVA*, pages 300–321, 2016.

[GPP+16]   Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *CCS*, pages 1626–1638, 2016.

[GZES17]   Berk Gülmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to violate web privacy with hardware performance events. In *ESORICS (2)*, pages 80–97, 2017.

[HR06]   Martin Hlavác and Tomás Rosa. Extended hidden number problem and its cryptanalytic applications. In *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 114–133. Springer, 2006.

[IAES15]   Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *IEEE Symposium on Security and Privacy*, pages 591–604, 2015.

[IEE00]   IEEE. Minutes from the IEEE P1363 working group for public-key cryptography standards, November 2000.

[IGI+16] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *CHES*, pages 368–388, 2016.

[Int09] International Organization for Standardization. Information technology - security techniques – cryptographic techniques based on elliptic curves. Part 5: Elliptic curve generation, 2009.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[KM05] Neal Koblitz and Alfred Menezes. Pairing-based cryptography at high security levels. In *Proceedings of Cryptography and Coding 2005, volume 3796 of LNCS*, pages 13–36. Springer-Verlag, 2005.

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, pages 549–564, 2016.

[LGS+17] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed JavaScript. In *ESORICS (2)*, pages 191–209, 2017.

[LSG+17] 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 *USENIX Security Symposium*, pages 557–574, 2017.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[MES17] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In *CT-RSA*, pages 21–44, 2017.

[MHMP13] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 435–452. Springer, 2013.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, pages 69–90, 2017.

[MN01] Atsuko Miyaji and Masaki Nakabayashi. New explicit conditions of elliptic curve traces for FR-Reduction. *IEEE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E84-A(5):1234–1243, May 2001.

[NS02] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Digital Signature Algorithm with partially known nonces. *J. Cryptology*, 15(3):151–176, 2002.

[NS03]   Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Des. Codes Cryptography*, 30(2):201–217, 2003.

[OKSK15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, pages 1406–1418. ACM, 2015.

[OST06]   Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and counter-measures: The case of AES. In *CT-RSA*, pages 1–20, 2006.

[Pag02]   Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive, Report 2002/169, 2002.

[Per05]   Colin Percival. Cache missing for fun and profit. In *BSDCon 2005*, Ottawa, CA, 2005.

[Sch87]   C. P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53(2-3):201–224, August 1987.

[Sch92]   Claus Peter Schnorr. *Block Korkin-Zolotarev bases and successive minima*. 1992.

[SCNS16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *AsiaCCS*, pages 317–328. ACM, 2016.

[SE94]   C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66(2):181–199, September 1994.

[SP17]   Raoul Strackx and Frank Piessens. The Heisenberg defense: Proactively defending SGX enclaves against page-table-based side-channel attacks. *arXiv preprint arXiv:1712.08519*, 2017.

[SPLI06] JH. Song, R. Poovendran, J. Lee, and T. Iwata. The AES-CMAC algorithm. RFC 4493, June 2006.

[SWG+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, pages 3–24, 2017.

[TSS+03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *CHES*, pages 62–76, 2003.

[TTMH02] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Hiyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *International Symposium on Information Theory and Its Applications*, Xi'an, CN, October 2002.

[VBWK+17] 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 *USENIX Security Symposium*. USENIX Association, 2017.

[vdPSY15] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *CT-RSA*, pages 3–21, 2015.

[WWB⁺17]   Shuai Wang, Wenhao Wang, Qinkun Bao, Pei Wang, XiaoFeng Wang, and Dinghao Wu. Binary code retrofitting and hardening using SGX. In *FEAST'17*, pages 43–49. ACM, 2017.

[XCP15]   Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (SP)*, pages 640–656. IEEE, 2015.

[XLCZ17]   Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *CCS*, pages 859–874, 2017.

[YGL⁺15]   Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. IACR Cryptology ePrint Archive, Report 2015/905, 2015.