SGAxe: How SGX Fails in Practice

Stephan van Schaik
University of Michigan
stephvs@umich.edu

Andrew Kwong
University of Michigan
ankwong@umich.edu

Daniel Genkin
University of Michigan
genkin@umich.edu

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

***Abstract*—Intel's Software Guard Extensions (SGX) promises an isolated execution environment, protected from all software running on the machine. A significant limitation of SGX is its lack of protection against side-channel attacks. In particular, recent works have shown that transient-execution attacks can leak arbitrary data from SGX, breaching SGX's confidentiality. However, less work has been done on the implications of such attacks on the SGX ecosystems.**

In this work we start from CacheOut, a recent confidentiality attack on SGX, which allows the retrieval of an enclave's memory contents. We show how CacheOut can be leveraged to compromise the confidentiality and the integrity of a victim enclave's long-term storage. By using the extended attack against the Intel-provided and signed architectural SGX enclaves, we retrieve the secret attestation key used for cryptographically proving the genuinity of enclaves over the network, allowing us to pass fake enclaves as genuine. Finally, we analyze the impact of our attack on two proposed SGX applications, the Signal communication app and Town Crier, an SGX-based blockchain application.

I. Introduction

Trusted Execution Environments (TEEs) are architectural extensions, recently introduced in commodity processors, which collectively provide strong security guarantees to software running in the presence of powerful adversaries. TEEs' promise to allow secure execution on adversary-controlled machines has spawned many new applications [57, 76, 84, 98, 100], both in academia [7, 8, 11, 18, 24, 27, 38, 78, 98] and industry [1, 2, 4, 22, 25, 39, 60, 63, 80]. Whereas several TEEs have been proposed (e.g., ARM's TrustZone [6, 70, 75] or AMD's Secure Encrypted Virtualization [53]), the currently prevailing x86 TEE implementation is Intel's Software Guard eXtensions (SGX) [20].

To support private and secure code execution, SGX provides isolated execution environments, called *enclaves*, which offer confidentiality and integrity guarantees to programs running inside them. While SGX's security properties strongly rely on the processor's hardware implementation, SGX removes all other system components (such as the memory hardware, the firmware, and the operating system) from the Trusted Computing Base (TCB). In particular, SGX ensures that the processor and memory states of an enclave are only accessible to the code running inside it and that they remain out of reach to all other enclaves and software, running at any privilege level, including a potentially malicious operating system (OS), and/or the hypervisor. At a high level, SGX achieves these strong security guarantees by encrypting enclave memory and protecting it with a secure authentication code, making the associated cryptographic keys inaccessible to software. Finally, SGX provides a *remote attestation* mechanism, which allows enclaves to prove to remote parties that they have been correctly initialized on a genuine (hence, presumed secure) Intel processor.

Notwithstanding its strong security guarantees, SGX does not protect against microarchitectural side channel attacks. As acknowledged by Intel, "SGX does not defend against this adversary" [46, Page 115] arguing that "preventing side channel attacks is a matter for the enclave developer" [45]. Starting with the controlled channel attack [94], numerous works have demonstrated side channel attacks on or from SGX enclaves [10, 21, 23, 32, 67, 91, 92]. Foreshadow [85] and following transient-execution attacks [17, 56, 79, 86, 87], demonstrated the capability of recovering contents from within the enclave's memory space.

Recognizing the danger, Intel released several patches, CPU microcode updates, and even new architectures designed to mitigate SGX side channel leakage via transient execution. However, the fix for TSX Asynchronous Abort (TAA) [44, 88], released in November 2019, was shown to be insufficient [79, 89, 90]. In particular, the CacheOut attack demonstrated the capability to leak SGX data by first evicting the data from the cache and then using TAA to recover it.

While CacheOut shows the ability to steal data from enclaves, it stops short of investigating the implications of this leak on the security of the SGX ecosystem. Given the increasing adoption of SGX as an alternative to heavyweight cryptographic protocols [2, 18, 31, 63, 98], there is clear danger in deploying SGX-based protocols on current Intel machines. With several SGX-based works aiming to offer some security guarantees even in the presence of information leakage, in this paper we ask the following questions:

What are the implications of information leakage from enclaves on the security of the SGX ecosystem? In particular, can SGX-based protocols maintain their strong security guarantees in the presence of side channel attacks?

A. Our Contributions

In this work we show that the security of the SGX ecosystem completely collapses, even in the presence of all currently-published countermeasures. Specifically, we test CacheOut on a fully updated machine, including a dedicated GPU to avoid Intel's Security Advisory SA-00219 [47], the latest SGX SDK and Intel-signed enclaves for mitigating LVI [86], and all of the countermeasures and microcode updates required to mitigate TAA [44, 79, 88], PlunderVolt [69], Zombieload [79], RIDL [87] and Foreshadow [85]. We show that despite activating all countermeasures, CacheOut [90] can effectively dump the contents of SGX enclaves.

Moreover, we show that CacheOut does not need to execute code within the enclave to dump its contents. Instead, it works while the enclave is completely idle. Thus, the attack bypasses all software-based SGX side-channel defenses, such as constant-time coding and countermeasures that rely on enclave code for ensuring protection [16, 26, 71, 77, 81].

Breaking the Integrity of Sealed Data. Next, going beyond attacks on SGX's confidentiality properties, in this work we extend CacheOut to also breach SGX enclaves' integrity. Specifically, we target the SGX *sealing* mechanism [5], which provides long-term storage of data, by securely generating an encryption key used to seal data by encrypting it before forwarding it to the operating system for long-term storage.

We use CacheOut to extract the sealing key from a victim enclave. The main challenge is that the SGX SDK implementation of the sealing API [40] wipes out the sealing key immediately after using it, leaving a short window in which the key can be retrieved. After recovering the sealing key, we use it to unseal and read the sealed information, then modify and reseal it. As SGX provides no integrity mechanism to detect such a change, the victim enclave now operates on data corrupted by the attacker.

Breaking Remote Attestation Next, we turn our attention to SGX's remote attestation protocol, which allows an enclave to prove to a remote party that it has been initialized correctly and is executing on a genuine (presumably secure) Intel processor. To attack attestation, we first use our attack on SGX's sealing mechanism and retrieve the sealing key of the SGX *Quoting Enclave*. Notably, unlike previous works [79, 86, 90] which build and sign their own quoting enclave, we attack a genuine, Intel-signed, production enclave, which employs all of Intel's countermeasures for transient-execution attacks, including the recent hardening for the LVI attack [86].

We then use the retrieved sealing key to unseal the persistent storage of the Quoting Enclave, which contains the private attestation key. We note that this attestation key is the only differentiating factor between a fully secure SGX enclave running on genuine Intel hardware and a malicious SGX simulator offering no security guarantees. Thus, we can build a malicious SGX simulator that passes Intel's entire remote attestation process. Performing remote attestation with the provider's server is usually the first task of any enclave running on the user's machine, resulting in a secure connection between the user's enclave and provider's servers. With the machine's production attestation keys compromised, any secrets provided by server are immediately readable by the client's untrusted host application, while all outputs allegedly produced by enclaves running on the client cannot be trusted for correctness. This effectively renders SGX-based DRM applications [35] useless, as any provisioned secret can be trivially recovered. Finally, our ability to read enclave contest and fully pass remote attestation also erodes trust in SGX-based secure remote computation protocols such as [1, 2, 7, 8, 11, 18, 22, 24, 27, 38, 39, 76, 78, 80, 98] as users cannot trust that their data will be properly protected by a genuine SGX enclave.

Exploiting SGX's Privacy Guarantees. Finally, we note that Intel's Extended Privacy ID (EPID) [52] severely compounds the consequences of compromised attestation keys. More specifically, when running in anonymous mode, EPID signatures are unlinkable, meaning that an attestation output cannot be linked to identity of the machine producing it [52]. This means that obtaining even a single EPID private key allows us to forge signatures for the entire EPID group, which contains millions of SGX-capable Intel CPUs. Thus, the leak of even a single key from a single compromised machine jeopardizes the trustworthiness of large parts of the SGX ecosystem. Luckily however, the unlinkable attestation mode is not recommended by default [52], allowing remote stakeholders to recognize our attestation quotes as coming from a different platform.

Undermining SGX-Based Protocols. To demonstrate the concrete implications of our breach of SGX, we conduct case studies examining how actual protocols, in both academic software and commercial products, may fail in practice when they rely upon SGX. We explore the Signal App [3] and Town Crier [98], which was acquired by ChainLink, a cryptocurrency with a \$1.5B market cap. We explore nuanced effects that breaching the integrity and confidentiality of SGX has on reliant protocols. This discussion is vital for securing the future of SGX-reliant protocols and designing such protocols to be resilient even when the underlying TEE is compromised.

Summary of Contributions. In this paper we make the following contributions:

- We use CacheOut to obtain the sealing key of enclaves, including architectural enclaves compiled and signed by Intel (Section III).
- We show how to use the sealing key of the Intel's Quoting enclave in order to retrieve the machine's attestation key (Section III-C).
- We show how to use the retrieved attestation key to forge SGX attestation quotes (Section IV).
- Finally, we discuss the implications of a broken attestation mechanism on Signal and on Town Crier (Section V).

B. *Current Status and Disclosure*

As part of the disclosure of CacheOut [90], we notified Intel of our findings in October 2019. While initial results about CacheOut's application to SGX were made public on a mutually agreed upon date of January 27th, 2020, Intel did not publish countermeasures mitigating CacheOut or the attacks described in this paper. Finally, Intel indicated that microcode updates mitigating the SGX leakage described in this paper will be published on June 9th, 2020.

C. *Threat Model*

Following the SGX threat model, we assume a compromised operating system where the attacker can install kernel modules or otherwise execute code with supervisor (ring-0) privileges. We assume that the hardware and software has been fully updated with Intel's latest microcode, and that the victim enclave contains no software bugs or vulnerabilities. Finally, we assume that the hardware is capable of supporting transaction memory (TSX). While this feature has been disabled by operating systems on all Intel machines released after 2018-Q4, we used elevated privileges in order to re-enable TSX RTM. We note that this does not violate the SGX threat model, as the OS is untrusted, while Intel's latest microcode update forcibly aborts TSX transactions during SGX operations [44].

II. Background

We now present background information on Intel's Software Guard Extensions and on microarchitectural attacks.

A. Intel Software Guard Extensions

Intel Software Guard Extensions (SGX) [5, 65] is an extension of the x86_64 instruction set, supporting secure execution of code in untrusted environments. SGX creates secure execution environments, called *enclaves*, which protect the code and data residing inside them from being maliciously inspected or modified. Additionally, SGX provides an ecosystem for remote attestation of enclaves' software and the hardware on which they run.

The SGX threat model assumes that the only trusted system components are the processor and Intel-provided and Intel-signed *architectural enclaves*. After booting, only the processor is trusted. The trust is extended to the *architectural enclaves* by hard-coding the public key used to sign them into the processor. Other than the *architectural enclaves*, SGX does not trust any software executing on the processor, including the operating system, the hypervisor, and the firmware (BIOS). The processor's microcode, however, is considered part of the processor and hence trusted.

For each enclave, SGX keeps an enclave-identity comprised of the enclave developer's identifier and a *measurement* representing the enclave's initial state. The developer's identifier, referred as MRSIGNER in SGX literature, is a cryptographic hash of the public RSA key the enclave developer used to sign the enclave's *measurement*. The enclave *measurement*, representing the enclave's initial state, is a cryptographic hash of those parts of the enclave's contents (code and data) that its developer chose to include in the measurement. The SDK implementation includes in the *measurement* all contents added to the enclave via `EADD`. Following the SGX nomenclature, we refer to this measurement as MRENCLAVE.

B. SGX's Sealing Mechanism

SGX provides enclaves with a mechanism for an encrypted and authenticated persistent storage. During CPU production, a randomly generated *Root Seal Key*, which is not kept in Intel's records, is fused in every SGX-enabled CPU. Using this key, the CPU can derive a sealing key, which can be used to encrypt and authenticate information from within the enclave's address space. Data that is *sealed* with this key, i.e., encrypted and authenticated, can be safely passed to the operating system for long-term storage, for example, on the computer's disk. SGX provides two types of sealing mechanisms, which we now describe (See [5, 49] for additional details).

Sealing Using the Enclave's Identity. As described in Section II-A, each enclave has a unique field, called MRENCLAVE, which is a cryptographic hash of the contents used to initialize the enclave code as well as some additional properties. Using the values of the Root Seal Key, MRENCLAVE, and the CPU Security Version Number (SVN) an enclave can use the `EGETKEY` instruction to derive a unique sealing key for sealing data before passing it to the operating system for long term storage. Note that as a consequence of this approach, for the same Root Sealing Key (i.e., on the same CPU), different software versions of the same enclave have different sealing keys. This prohibits both data migration between different versions of enclaves as well as using these sealed keys for intra-enclave communication.

Sealing Using the Developer's Identity. An alternative option to the one discussed above is to generate the sealing key using the Root Seal Key, the SVN, and MRSIGNER (instead of MRENCLAVE). As explained in Section II-A, MRSIGNER is a cryptographic hash of the public RSA key of the enclave developer and there remains the same for all enclaves developed by the same vendor. Thus, data sealed in this way is accessible by different versions of the same enclave, as well as by different enclaves belonging to the same vendor.

C. Caching Hierarchy in Modern Processors

Modern processors contain a series of caching components that all collaborate to bridge the speed gap between the fast processor core and the slower memory, see Figure 1.

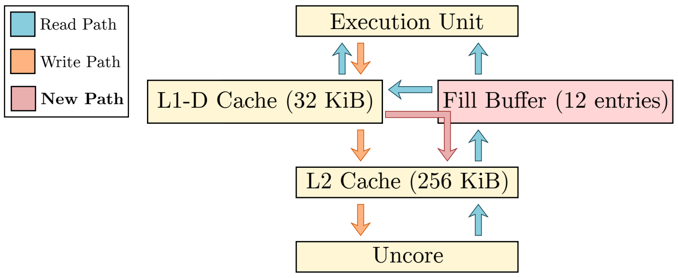

Fig. 1: Caching hierarchy in Intel processors.

When accessing memory, the processor first searches if the data exists in the L1-Data cache. In case of a cache hit, when the data is found in the cache, the accessing instruction receives the data from the cache and proceeds to process it. Conversely, in case of a cache miss, when the data is not found in the cache, the processor resumes its search further down the memory hierarchy until the data is found. When data is retrieved from lower-level caches, it is often stored in higher-level caches for possible future use.

Out-of Order Execution and Line Fill Buffers. To exploit the inherent concurrency in software, the execution engine of modern processors is out-of-order. That is, rather than executing instructions in the order that is stipulated by the program, the processor executes instructions as soon as the data they process is available. Consequently, multiple instructions may be in various stages of execution at the same time. The main use of the line fill buffers (LFBs) is to avoid locking the cache during cache-miss handling, in order to allow concurrently-executing instructions to access the cache. Specifically, when a cache miss occurs, the processor allocates an entry in the LFB, which processes cache misses asynchronously. When the data arrives from the lower levels of the hierarchy, the LFB transfers the retrieved data to the waiting instruction and copies it into the cache for potential future use.

Cache Evictions. The capacity of the L1-Data cache is typically relatively small (32 KiB in Intel processors). Hence, when new data is being cached, old data has to be evicted from the cache. In case the old data has not been modified, it can just be discarded. However, if the data to be evicted has been previously modified, the processor needs to percolate the modifications in lower levels of the cache hierarchy, and ultimately in the memory.

D. Microarchitectural Attacks

Cache Attacks. Because the behavior and the state of microarchitectural components affects the execution speed of programs, programs sharing the use of components can detect contention and infer information on the execution of other programs [28]. A large body of work exploits contention on space in caches to recover cryptographic keys [13, 29, 30, 50, 59, 66, 73, 74, 96, 99] and other sensitive information [10, 33, 37, 95]. Temporal contention, i.e., exploiting the limited capacity of the cache to handle concurrent accesses, can also leak information [97]. Finally, cache contention can be used to construct covert communication channels that bypass the system's protection guarantees [59, 64].

Transient Execution Attacks. Starting form Spectre and Meltdown [55, 58], there has been a large body of work on exploiting speculative and out-of-order execution to leak information (see [15] for a survey). Specifically, in out-of-order execution, the processor aims to predict the future instruction stream in order to optimize the execution of future instructions. However, when the processor mispredicts the instruction stream it may execute instructions that deviate from the program's nominal behavior, bypassing the program's control flow, or permission checks. When the processor discovers the misprediction, it attempts to revert the incorrect execution by discarding the outcome of misspredicted instructions, never committing them to the processor's architectural state. However, as discovered by Spectre and Meltdown [55, 58], changes performed by incorrect transient execution do affect the processor's microarchitectural state, allowing attackers to use the CPU's microarchitecture as a means of storing information. More specifically, in transient-execution attacks, the attacker first causes transient execution of instructions that access data not normally used by the program. The attacker then uses a microarchitectural covert channel to exfiltrate the data, thus bypassing the processor's state reversion.

Meltdown-Type Attacks. While other transient attacks do exist [9, 36, 51, 54, 55, 56, 61, 62], Meltdown-type attacks [15] exploit delayed exception handling on Intel processors in order to bypass hardware-backed security boundaries. Following their original utilization in Meltdown [58] for reading kernel data, such attacks have been used to recover the content of floating-point registers [82, 89] as well as the L1-Data cache [85, 93]. More recently, a new type of transient execution attacks was discovered, where the attacker is able to leak information from internal microarchitectural buffers, including the line fill buffers [79, 87], write combining operations [68], and store buffers [14]. Dubbed Microarchitectural Data Sampling (MDS) attacks by Intel, these attacks are likened to "drinking from the firehose", as the attacker has little control over what data is observed and from what origin.

Recognizing the danger stemming from uncontrolled data leaking from various internal microarchitectural data structures, two high level defense strategies have emerged. First, for older processors, Intel have issues microcode updates that overwrites the contents of various microarchitectural elements such as internal buffers or L1-D cache [41, 43]. These were then adopted by all major operating systems, flushing microarchitectural data structures on security domain changes. In parallel, Intel has launched new processor architectures, which attempt to mitigate transient execution issues in hardware.

E. CacheOut and LVI: Ineffective Flushing

While secure at first sight, recent works [86, 90] have shown that Intel's contents overwriting strategy does not fully mitigate transient execution issues.

Load Value Injection (LVI). More specifically, LVI [86] shows how an attacker can use the still leaky buffers in order to *inject* values into the address space of a victim process or SGX enclave. As the victim now performs transient execution on incorrect data, it possible to temporary hijack the victim's control flow, or expose his secrets. Since LVI is particularly effective against SGX enclaves, Intel has issued a series of software mitigations [42], hardening enclave code and data against injection attacks by restricting speculation.

CacheOut: Leaking Data via Cache Evictions. Next, buffer overwriting does offer any protection in case the attacker can somehow move data back after the buffer's contents was overwritten. Indeed, utilizing the observation that when modified data is evicted from the L1-Data cache it can sometimes end up in the LFB, the CacheOut [90] attack was able to successfully bypass Intel's buffer overwriting countermeasures, again breaching the confidentiality of nearly all secuirty domains.

Figure 2 is a schematic overview of the two scenarios considered in CacheOut [90], where an attacker can observe the victim's data through the fill buffer (despite Intel's buffer overwriting countermeasures). The left shows how an attacker can sample data that the victim is reading. Here, the the attacker first ensures that the data the victim is about to read is

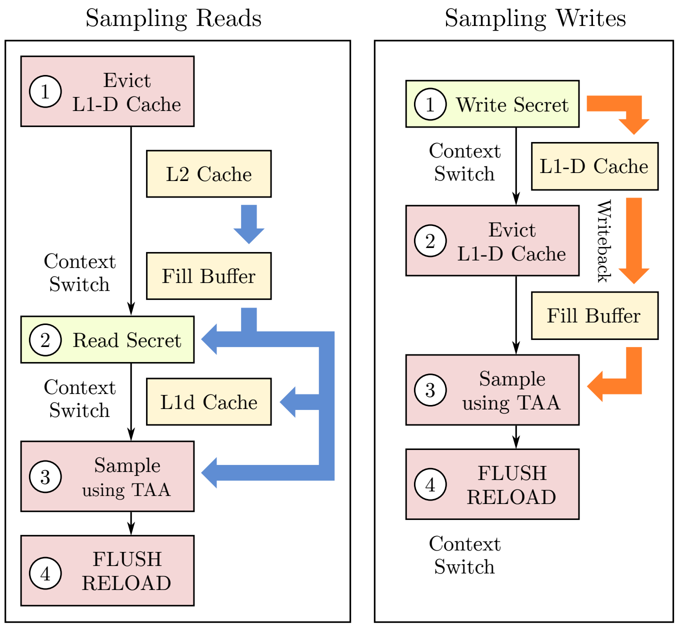

Fig. 2: Overview of how the victim's data can end up in the LFB allowing an attacker to sample the data from the Fill Buffers using an attack like TAA. Victim activity, attacker activity, and microarchitectural effects are shown in green, red, and yellow respectively. The context switches both illustrate the OS flushing the MDS buffers before switching to the other process as well as switching between actual Hyper Threads. (from CacheOut [90])

no longer in the L1-D cache. The attacker does this by filling up the L1-D cache with its own data to evict the victim's data. Now when the victim executes, the victim tries to read some data again. However, since the data that the victim wishes to read is no longer present in the L1-D cache, the CPU now has to fetch the data from the higher-level caches or the physical memory. Since cache misses have to be served through the LFB, the attacker now has an opportunity to sample the data from the LFB using an attack like TAA [44].

On the right we see how an attacker can sample data from the victim when writing data. In this case, the victim will write the data to the L1-D cache, as the CPU will handle updating the higher-level caches and the physical memory in the background. This presents an opportunity to the attacker to leak the data. Similar to the case on the left, the attacker fills up the L1-D cache to evict the victim's modified cache line. This forces the CPU to update the higher-level caches immediately, as there is no longer space available in the L1-D cache to host the victim's data. In order to perform this update, the data has to pass through the LFB. Now that the attacker forcefully pushed the victim's data into the LFB, it can again use an attack like TAA to sample the data from the LFB.

In both cases the attacker can fully manipulate the micro-architectural state to ensure that reads or writes originates from the victim will have to pass through the LFB. Moreover, in the case of writes, an attacker can force the data into the LFB and sample it immediately without the intervention of a context switch. This renders mitigations that rely on flushing these leaky buffers during context switches ineffective.

Leaking SGX Enclaves via CacheOut. CacheOut further demonstrates the use of the attack to build a read primitive that leaks information from SGX enclaves. The read primitive uses a modified SGX driver that exercises the control the operating system has on the enclave to provide an interface that swaps enclave pages out and in. In a nutshell, SGX includes two instructions, `ewb` and `eldu`, which are used to export memory pages from an enclave, while encrypting their contents, and to import such encrypted pages back. Swapping a page out and in leaves the contents of the page in the L1-Data cache, and because the processor writes the contents into the enclave, these contents are marked as 'modified'. Concurrently with swapping the pages, CacheOut evicts data from the L1-Data cache, and uses the TSX Asynchronous Abort technique [44] to leak the evicted data from the LFB. Using this technique, an attacker can target specific bytes in the page that is being swapped out and in, and retrieve them.

III. Attacking SGX's Sealing Mechanism

The CacheOut attack described in Section II-E is capable of breaching SGX's confidentiality guarantees, effectively dumping the contents of any SGX enclave available on the target machine. It cannot, however, breach SGX's integrity guarantees as it is unable to modify the contents of the enclave's code or data.

In this section, we show an attack against SGX's sealing mechanism, which is a mechanism designed to provide enclaves with encrypted and authenticated persistent storage. In a nutshell, we use CacheOut to recover the sealing keys from within the address space of Intel's production quoting enclave. Finally, we use the recovered sealing keys in order to decrypt the long term storage of the quoting enclave, obtaining the machines EPID attestation keys.

A. *Extracting SGX Sealing Keys*

Key derivation using `EGETKEY` leaves the sealing key in the memory of the victim enclave. Thus, in principle, it is possible to read this key using CacheOut [90]. However, immediately after using it to encrypt or decrypt the sealed data, the implementation of SGX's sealing API erases the sealing key from memory. Hence, to extract the key we need a method for launching CacheOut during the data sealing or unsealing process, before these keys are erased from the memory.

Tracing Quote Enclave Execution. We use a variant of the controlled-channel attack [94] to induce an Asynchronous Enclave Exit (AEX) when the enclave calls our functions of interest. More specifically, we use `mprotect()` using the `PROT_NONE` flag to unmap the page that contains our function of interest. When the enclave then tries to execute the function, it instead results in an AEX and a page fault, returning control back to the operating system (or, more specifically, to the attacker-controlled signal handler). However, since pages are 4 KiB in size, they may host multiple functions, meaning that we have to carefully select the order of pages to unmap to properly stop the quoting enclave at the point while the sealing keys are still in memory. To achieve that, when control returns back to our malicious signal handler, we mark the current page as executable and proceed to unmap the page of the next function of interest. Finally, we return control flow back to SGX, in order to resume enclave execution until the next function of interest.

Using the above describe method, we can trace the execution of the quoting enclave until a point after the sealing keys have been stored in memory but before they have been zeroed out. Once the quoting enclave execution has reached this point, we never resume its execution, proceeding to the next phase of our attack.

Leaking Enclave Contents. Following the method of Cachout [90], we use a modified SGX Linux driver that repeatedly swaps the page holding the sealing keys in and out of the physical memory, using SGX's `ewb` and `eldu` instructions with a short delay in between. As observed by [90], this physical memory activity requires SGX pages to be repeatedly encrypted and decrypted. Therefore, the CPU's L1-D cache will contain a decrypted copy of the page's contents. Next, using another Hyper-Thread on the same physical core, the attacker uses CacheOut's methodology, and evicts the cache set whose values he would like to leak from the L1-D cache to the CPU's LFB. Finally, the attacker uses TAA to recover the values from the CPU's LFB. Finally, note that since the victim enclave never resumes execution, no software mitigation can be used to prevent the information from leaking.

Pinpointing Functions of Interest. The above description assumes that the attacker knows the precise execution sequence of functions inside the quoting enclave from the `get_quote()` function to the AES decryption operation used to unseal the EPID keys. To recover this calling sequence, we first compiled our own version of the quoting enclave using Intel's SDK and PSW. We then executed the quoting enclave in debug mode with dummy inputs, tracking execution up to the `sgx_rijndael128GCM_decrypt()` function, which is used to decrypt the (dummy) EPID blob. The `sgx_rijndael128GCM_decrypt()` function proceeds by calling `l9_aes128_KeyExpansion_NI()` to perform AES key scheduling, expanding the AES sealing key into individual AES round keys. Next, in order to stop enclave execution immediately after the key scheduling, we leverage the fact that the recently-introduced LVI countermeasures [42] mean that enclaved functions no longer return using the `ret` instruction directly, but instead call `__x86_return_thunk()` to return from a function. Thus, by unmapping the page containing `__x86_return_thunk()` just after the execution of of `l9_aes128_KeyExpansion_NI()`, we can run the quoting enclave until the end of the AES key expansion process, and stop its execution immediately after it, while the round keys are still present in the enclave's memory. Experimentally verifying this, we were able to recover some round keys from this self-compiled and signed quoting enclave, verifying the ground truth using a debugger.

Attacking Intel's Production Quoting Enclave. After our practice run with our self-compiled quoting, we proceeded to attack SGX's official quoting enclave, as compiled and signed by Intel Unfortunately, the binary supplied by Intel lacks the debugging symbols from before, and the debugger was not able to insert breakpoints or dump memory. Therefore, we first stripped Intel's quoting enclave from its signature, and ran it in debugging mode using our own key and dummy inputs. We then located the functions of interest required to implement the above unmap-execute-remap strategy, by using a debugger and comparing the assembly of Intel's quoting enclave with our self-compiled version. After determining the memory address of the required function, we implemented the above strategy blindly (i.e., without the aid of the debugger) against the production quoting enclave.

B. Empirical Evaluation

Experimental Setup. The experiments performed in this section were conducted using an Intel Core i9-9900K (Coffee Lake Refresh) CPU, running Linux Ubuntu 18.04.4 LTS with a 5.3.0-53 generic kernel and microcode update `0xcc` (the latest one released by Intel at the time of writing). We have used Intel's SGX SDK version 2.9.100.2 and version 2.9.100.1 of the Intel Quoting Enclave, which is the latest at the time or writing and includes software hardening against the recent LVI transient execution attack [86].

As the Coffee Lake Refresh architecture has hardware mitigations against Meltdown, Foreshadow/L1TF, and MDS, Hyper-Threading is considered safe and is therefore left enabled. We have also re-enabled TSX support (which is off by default on these machines). However, since SGX operations force TSX aborts [44], this configuration is also considered safe by Intel for running SGX enclaves. Since the integrated Intel GPU is considered to be insecure [47], we installed a dedicated GPU, disabled the integrated GPU and enabled PlunderVolt mitigations [69]. Finally, we have verified that the system was indeed in a safe configuration using Intel's remote attestation example[*], which outputted `Enclave TRUSTED` from Intel's Attestation Server (IAS). This implies that Intel considers our configuration as trustworthy and secure for SGX operations.

Key Extraction. We ran the production quoting enclave five times, each time using the above-described strategy to stop its execution precisely after the generation of the AES round keys corresponding to the EPID sealing key. Next, using our technique to leak enclave contents, we have attempted to read the memory locations corresponding to the AES round keys. Out of the 176 bytes corresponding to round keys, we found that on average 148.6 bytes (or about 84%) were correctly recovered across our five attempts.[†] However, as we do not know which key bytes are wrong, we resort to algorithmic methods for recovering AES keys from noisy information which we now describe.

[*]https://github.com/intel/sgx-ra-sample

[†]These statistics were later computed once the corresponding sealing key was successfully recovered.

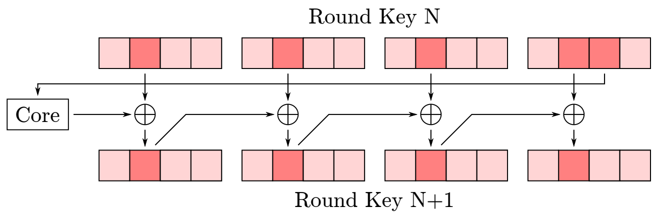

Fig. 3: In the 128-bit AES key schedule, four bytes from a round key are determined by five bytes of the previous round key.

C. EPID Key Recovery

Before describing our method for recovering the AES sealing key from the noisy AES round keys we extracted in Section III-A, we present an overview of AES's key scheduling algorithms.

AES Key Scheduling. As 128-bit AES uses 10 encryption rounds, the key scheduling algorithm generates 10 round keys where each round key is four 32-bit words. Each round key is computed either by XOR'ing a word from the previous round key (or the initial key) with the previous word in the current round key; or by performing the key schedule core which rotates the bytes in a word and then maps each byte to a new valuea and XOR'ing the result together with another word from the previous round key. See Figure 3 for an outline of how to generate round key $n + 1$ from round key n.

Next, consider the derivation of round key $n+1$ from round key n. To compute each byte in round key $n + 1$, we only need two bytes, one from the previous 32-bit word and one from round key n (Figure 3). Thus, as initially observed by Halderman et al. [34], it is possible to detect an correct errors in the AES round keys, by ensuring that each byte is consistent with the values of the two bytes used to compute it.

Recovering the Sealing Key. To recover the AES sealing of SGX's quoting enclave from the data obtained in Section III-A, we used a technique similar to [34]. More specifically, the information we collected using our attack contains multiple candidates for each byte offset. Next, inspecting the layout of the binary of the SGX quoting enclave compiled and signed by Intel, we are able to deduce the exact mapping between addresses and round keys.

Observing the data collected across the five attestation runs performed in Section III-B, for each round key $1 \le n \le 10$ and for each byte, we assume that the top-1 candidate is indeed the correct value for the byte, and proceed to compute a candidate for the next round key $(n + 1)$ as illustrated in Figure 3. Once we have computed the candidate round key $n+1$, we can check every byte of the candidate for the $n+1$ round against the observed leakage data. Furthermore, since the AES key schedule is a reversible operation, given any two out of three bytes, that is two for the input and one for the output, we can compute what the third byte should be and check if the result is among the candidates. We use this technique to filter out the candidates that are part of an AES key schedule from our traces and are able to observe multiple complete AES round keys.

Finally, we note that it is possible to compute all of the AES round keys, and the original encryption key, any given AES round key. Thus, using the above-described technique, we were able to recover the sealing key for Intel's quoting enclave from the data obtained in Section III-B within seconds using a regular laptop computer.

Unsealing the Machine's EPID Attestation Key. Having successfully recovered the AES sealing key of the machine's quoting enclave, we proceeded to unseal the EPID key blob. Upon decrypting the blob and verifying the corresponding authentication tag, we were able to obtain the machine's private EPID keys used for SGX attestation in both linkable and unlinkable modes.

IV. Attacking SGX Attestation

One of most compelling integrity properties that SGX provides is the ability of an enclave to attest to a remote verifying party that it is running on genuine Intel hardware and not on an SGX simulator. This attestation process proves to the remote party that the enclave leverages the data confidentiality and execution integrity properties provided by SGX and, therefore, the remote party can transfer secret data to the enclave while being assured that the enclave will not intentionally leak the secret data. Finally, attestation also allows the remote party to trust that the enclave's outputs are the result of a trustworthy correct execution, as opposed to being crafted via a malicious simulator.

Having recovered the machine's EPID attestation keys in Section III, in this section we show how to construct a malicious simulator which passes attestation as if it was an SGX enclave running on a genuine Intel processor, while executing the enclave code outside of an actual enclave, in an arbitrary (and possibly incorrect or leaky) ways. Finally, as the private attestation keys are all that distinguish genuine Intel hardware from potentially malicious simulators, the remote verifying party has no way of distinguishing between the two and thus cannot trust the computation's output to be correct.

Before describing our attack, we now provide some background about SGX's provisioning, quoting, and attestation processes.

A. SGX Remote Attestation

The remote attestation process allows a *remote* verifying party to verify that a *specific* software is correctly initialized and executes within an enclave, on a genuine Intel CPU. At a high level, this is performed as follows (see [52] for an extended discussion).

In addition to the Root Sealing Key (Section II-B), every SGX-enabled CPU is also shipped with a randomly-generated Root Provisioning Key (Step 1, Figure 4). However, unlike the Root Sealing key, Intel does retain a copy of the Root Provisioning Key, as it acts as a shared secret between Intel and every individual CPU. Next, Intel provides two special

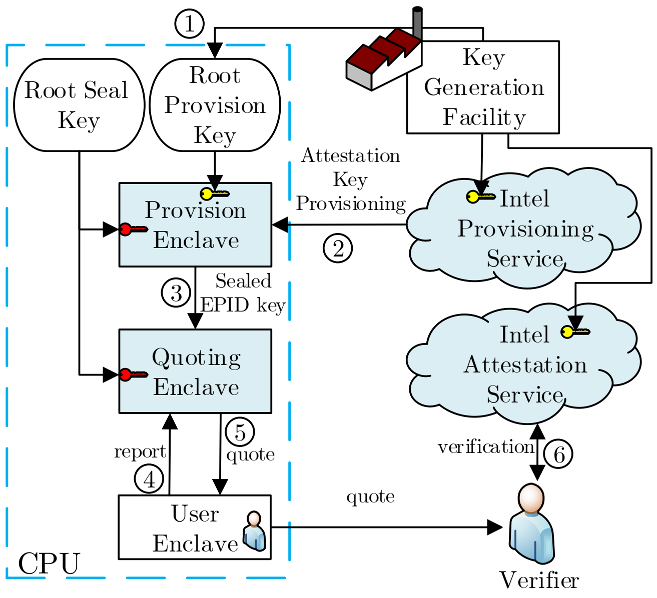

Fig. 4: SGX's Attestation Process.

enclaves, called the *Quoting Enclave* and the *Provisioning Enclave* which are used in the attestation process.

Attestation Key Provisioning. The initialization phase of the SGX attestation protocol consists of the Provisioning Enclave contacting Intel's provisioning server, transmitting the CPU's provisioning ID, and claimed Security Version Number (SVN). As the provisioning ID uniquely identifies a specific CPU, it is only accessible to the Intel-signed Provisioning Enclave and is sent encrypted to the provisioning server under Intel's public key. After recovering the root provisioning key, corresponding to the CPU's provisioning ID, the provisioning server and Provisioning Enclave proceed to execute the *Join* phase of Intel's Enhanced Privacy ID (EPID) protocol [12], using the root provisioning key as a shared secret (Step 2, Figure 4).

At a high level, Intel's EPID protocol is a type of group signature that allows a CPU to sign messages (using its private signing keys) without uniquely disclosing its identity. When executed in unlinkable mode, all that an external observer (e.g., Intel) can do is to verify the signature (thereby becoming convinced that it was signed by a genuine Intel CPU belonging to the group), without being able to link it to any specific Intel CPU or to other signatures it previously signed. See [12] for additional discussion.

Sealing the EPID Key. The *Join* phase of the EPID protocol results in the Provisioning Enclave obtaining a private EPID signing key, which is not known to Intel. The Provisioning Enclave then generates a sealing key for sealing the EPID signing key, using the CPU's Root Sealing key, its SVN and the MRSIGNER value of the Provisioning Enclave. It then seals the private EPID key using this sealing key and outputs it to the OS for long term storage (Step 3, Figure 4). Notice that as the Provisioning Enclave is provided and signed by Intel, its MRSIGNER value is a hash of Intel's public key. Consequently, any Intel-signed enclave can unseal the CPU's private EPID key by regenerating the sealing key used to seal it. While this design feature is indeed useful, as it allows the Quoting Enclave (also signed by Intel) to unseal the private EPID key, it is also dangerous as the OS actually has an encrypted copy of the CPU's private EPID keys.

Local Attestation. When an enclave wants to prove to a remote verifier that it is running on genuine Intel hardware with a specific security version, it first needs to prove its identity to the *Quoting Enclave*, which is another special enclave provided and signed by Intel, via a process referred to by Intel as *local attestation* [5, 49]. At a high level, this is done by having the proving enclave use the `ereport` instruction, which prepares a report containing the MRENCLAVE and MRSIGNER values of the proving enclave. The report is also signed using a key that is only accessible to the Quoting Enclave. The proving enclave then passes the report to the Quoting Enclave, which proceeds with the remote attestation process (Step 4, Figure 4).

Remote Attestation. Upon receiving the report from the proving enclave, the Quoting Enclave performs the remote attestation process which we now describe. Indeed, after verifying that the report was correctly signed by the `ereport` instruction, the Quoting Enclave proceeds with unsealing the EPID private key that was originally sealed by the Provisioning Enclave. Recall that the EPID private key was sealed using a sealing key derived from the CPU's SVN version, Root Sealing Key, and the MRSIGNER value of the Provisioning Enclave. As both the Quoting Enclave and the Provisioning Enclave are signed by Intel (and thus have the same MRSIGNER value), the Quoting Enclave can regenerate this sealing key and subsequently unseal the private EPID key. Next, using the unsealed private EPID signing key, the Quoting Enclave executes the *Sign* phase of the EPID protocol and signs the report given to it by the proving enclave, creating an *attestation quote*. Finally, the Quoting Enclave returns the quote to the proving enclave, which in turn forwards it to the remote verifying party (Step 5, Figure 4).

Attestation Verification. After the proving enclave sends the signed quote to the remote verifying party, the remote party interacts with Intel's Attestation Server (IAS [48]) and provides it with the *quote* it obtained from the Quoting Enclave (Step 6, Figure 4). Next, IAS performs the *Verify* phase of the EPID protocol while ensuring that the signer's private EPID key has not been revoked by Intel. Intel's server completes the attestation by sending its response (`OK`, `SIGNATURE_-INVALID`, etc.) to the remote party. The server's response also contains the quote itself and is signed with Intel's signing key, generating a signed attestation transcript which can later be verified by any party that trusts Intel's public key.

B. Breaking SGX Attestation

In this section, we describe our attack on SGX's attestation protocol. As explained above, the Quoting Enclave, which can access the EPID signing keys, will not sign a local attestation report without first verifying it. Moreover, as mentioned in

Section IV-A above, the operating system actually has a copy of the EPID private keys, which are sealed by a key derived from the CPU's Root Sealing Key. Our attack thus proceeds as follows.

Step 1: Recovering the Sealing Keys. Using the attack described in Section III on the Quoting Enclave, our attack recovers the sealing keys used for sealing the EPID signing keys.

Step 2: Unsealing the EPID Signing Keys. With the above sealing keys, our attack proceeds to unseal the private EPID keys, originally sealed by the Provisioning Enclave.

Step 3: A Malicious Quoting Enclave. Using the source code of Intel's quoting enclave [40], we have constructed a malicious quoting enclave that signs *any* local attestation report with the EPID keys, obtained in Step 2 above, without first verifying it.

Step 4: Breaking Attestation. Consider a malicious software that would like to masquerade as a specific enclave and prove its "authenticity" and SGX security properties via remote attestation. Given an enclave to masquerade, the malicious software first generates a *false* local attestation report with the values of MRENCLAVE and MRSIGNER corresponding to the enclave it wants to masquerade, as well as other metadata required for generating the local attestation report. It then sends this report to our malicious quoting enclave.

We notice here that the malicious software is unable to sign the *local* attestation report, as it doesn't have access to the appropriate signing key. However, as our malicious quoting enclave does not verify the report, the report does not have to be signed. Next, using the unsealed EPID keys, our malicious quoting enclave generates an attestation quote by signing the local (false) attestation report, which is then sent to the remote party. Finally, the remote verifying party attempts to verify the malicious quote using Intel's Attestation Server (IAS). As the quote was indeed correctly signed by the malicious quoting enclave using genuine and non-revoked EPID keys, Intel's attestation server will accept the malicious quote and generate a signed transcript of the response. The transcript falsely convinces the remote party that the enclave is running on a genuine Intel CPU, which is designed to provide confidentiality and integrity, while it is actually running under the attacker's control, outside of SGX, and thus does not offer any security guarantees.

C. Empirical Evaluation

In this section, we empirically demonstrate the feasibility of our attack on SGX's attestation mechanism.

Extracting EPID Keys. As mentioned in Section III-C, we have successfully extracted the EPID sealing keys from a genuine SGX quoting enclave and subsequently unsealed the machine's private EPID keys. After unsealing the machine's private EPID key, we modify our malicious quoting enclave to directly provide the decrypted EPID blob as well as the additional MAC text. We confirmed that our malicious quoting enclave functions correctly by first testing the `get_quote()` call to see if we can successfully sign our reports. Next, we confirmed the correctness by using Intel's example of how to perform Remote Attestation using SGX and running it with our malicious quoting enclave. Finally, we performed the same test using our malicious quoting enclave on a Whiskey Lake machine that was not able to pass remote attestation due to an issue with its integrated GPU [47], which now appears to be trusted by Intel as it claims to be the Coffee Lake Refresh machine from Section III.

Signing Fake Attestation Quotes. Demonstrating our ability to sign arbitrary attestation quotes, we created a local attestation report setting the MRENCLAVE field, "representing" the SHA-256 of the enclave's initial state, to be the string "*This enclave should not be trusted*", the MRSIGNER, "representing" the SHA-256 of the public key of the enclave writer, to be "*SGAxe: How SGX Fails in Practice*", and the report's debug flag to 0, thereby indicating that the enclave is a production enclave. We have also populated the report's body (commonly used for establishing a Diffie-Hellman key exchange with the enclave corresponding to the report) to be "*Mary had a little lamb, Little lamb, little lamb, Mary had a....*". Finally, we signed the report using our malicious quoting enclave using our extracted EPID keys, thereby producing an attestation quote.

Quote Verification. To verify the validity of our quote, we contacted Intel's Attestation Server (IAS) and provided it with the above generated quote. As explained in [12, 52], the attestation server will only approve the quote if it can verify that the quote's EPID signature is correct. Since we have correctly extracted a non-revoked EPID private signing key, using version 3 of the attestation API [48], the attestation server deemed our quote as correct and replied with "`isvEnclaveQuoteStatus: OK`". The IAS also signed its response with Intel's private key and accompanied it with the appropriate certificate chain leading to Intel's CA certificate.

Attestation Protocol Versions. While Intel's official remote attestation example uses version 3[‡], the documentation of the IAS API [48] does mention a recently introduced support of version 4. At a high level, the difference between the two versions revolves around the use of LVI [86] software hardening countermeasures [42]. As Intel has no way of remotely verifying if such hardening was perform or not, on machines vulnerable to LVI (nearly all currently available Intel machines), version 4 of the attestation protocol outputs SW_-HARDENING_NEEDED which indicates that our malicious quoting enclave's "has been verified correctly but due to certain issues affecting the platform, additional SW Hardening in the attesting SGX enclaves may be needed" [48, Page 22]. Intel further instructs "the relying party should evaluate...whether the attesting enclave employs adequate software hardening to mitigate the risk". This indicates that Intel asks enclave providers to trust their own enclaves only if they have performed LVI mitigations [42], without being able to remotely verify so. Using version 3 of the attestation however, removes

[‡]https://github.com/intel/sgx-ra-sample

this message, results on an "`isvEnclaveQuoteStatus: OK`" status, meaning that the enclave is fully trusted. In a deployment scenario, we expect most SGX enclaves to be hardened against LVI attacks, having their vendors either use version 3 of the attestation protocol, or treat the SW_HARDENING_NEEDED message of version 4 as OK (as already done by Signal[§]).

V. Subverting SGX-Based Protocols

We will now proceed to explore the implications of compromising one of SGX's attestation keys, as demonstrated in Section IV. A number of recent works, both academic [7, 8, 11, 18, 24, 27, 38, 78, 78, 98] and otherwise [1, 2, 4, 22, 25, 39, 60, 63, 80], have begun to incorporate SGX into the design of their systems. If this is not done with careful consideration of what guarantees SGX actually provides, however, reliance upon SGX actually weakens the system. The three following case studies serve to highlight how misplaced trust in SGX can subvert the security of an entire system.

A. Subverting Town Crier

A long standing problem facing the adoption of smart contracts [83] is the difficulty of bridging the gap between contracts operating on the blockchain and real world data. Town Crier [98], which was recently acquired by ChainLink, a cryptocurrency with over $1.5 billion in market capital, aims to address this within the Ethereum blockchain by providing authenticated data oracles. By combining an on-chain smart contract frontend with an SGX enabled, off-chain backend, Town crier can reliably fetch data from HTTPS enabled websites onto the chain. By using SGX as a blackbox, and assuming that its security guarantees hold, Town Crier leverages SGX to guarantee the correctness of the data that is fetched onto the chain. Assuring that the data is correct and protected from tampering is critically important, as financial derivatives may potentially rely upon said data, thereby giving malicious entities a strong financial incentive to modify the data in their favor.

Data Flow. The Town Crier data flow is as follows: reliant contracts request data from these oracles by sending messages to the oracle's smart contract front-end. The back-end runs within an enclave, and fetches the requested data. The data is then delivered to the chain by sending a message from a wallet whose private ECDSA key is known only by the enclave; clients must verify the SGX attestation of the enclave's code and the public key of its associated wallet. Assuming the correctness of SGX, this guarantees that messages delivered from the corresponding wallet originated from the enclave, whose code has been attested to.

Restricting SGX Compromise. While the implication of breaching the integrity of the SGX enclave is straightforward (the compromised enclave can report incorrect arbitrary answers to queries), Town Crier aims to hedge against the compromise of a single SGX instance via replication. This is potentially complicated, however, by the anonymity provided by Intel's EPID signature scheme.

Intel's EPID signature scheme can be run with either a fully anonymous attestation policy, or a pseudonymous policy. In the former, EPID signatures are completely unlinkable, meaning that an attestation cannot be linked to the identity of the machine that produced it. This means that the attacker can leverage the attestation key obtained by compromising just a single SGX machine to forge an arbitrary number of attestations for Town Crier enclaves, all compromised by the attacker, but seemingly running on different machines.

This makes it difficult to rely on replication for security against SGX compromise; relying contracts and clients would need to be especially judicious about which SGX enclaves to trust, which defeats the purpose of using SGX to remove trust from the data oracle operator. As an example, an adversary who has compromised a single SGX instance can create a seemingly benign Town Crier contract that takes a majority vote from numerous seemingly separate SGX machines. Since the attacker can forge signatures for each "separate" SGX machine, the adversary has complete control over the data that the contract receives and distributes.

B. Subverting Signal's Private Contact Discovery

The Signal messenger app is a popular platform for sending end-to-end encrypted messages between smart phone contacts. To facilitate this, Signal plans to use SGX for Private Contact Discovery [63] (discussed in this section) as well as for Secure Value Recovery [4, 60] (discussed in Section V-C).

Building a Social Graph. Signal builds its social graph by piggybacking upon the network composed of users' contacts that they already store on their phones. This means that when a user joins Signal, she needs to discover which of her contacts are also on Signal before she can begin to contact them. Naively, this can be accomplished by having users submit hashes of their contacts to the Signal server, with the server responding by indicating which contacts are registered with them.

This process, however, can potentially expose a user's contacts to the Signal server, as a dictionary attack against the hashed passwords is computationally feasible, given the small number of possible pre-images. This can be undesirable if users do not trust the server, either because the operator's themselves are untrustworthy, or if the user is concerned that the server's database can be compromised by a malicious third party (e.g. through a successful hacking attempt or a subpoena).

SGX for Contact Discovery. To address this, Signal has proposed moving this contact discovery service to within an SGX enclave [63]. Users would send their contacts to the enclave, which would then match them against the set of registered users and return the intersection. To assure the confidentiality of the users' contact list, users would first need to verify via remote attestation that the enclave is executing the correct code. To ensure that the server learns nothing about

[§]https://github.com/signalapp/sgx_common/pull/1#issue-395160348 the clients' contact books, Signal also designed their contact discovery service to use an oblivious hash-table construction to protect against cache side-channel attacks on the enclave.

Leaking Contacts. By completely breaching SGX in the manner described in Section IV, a malicious Signal server would be able to create an enclave that exposes all of the data it receives, while at the same time proving to clients via remote attestation that the enclave is performing the benign, published operations of private contact discovery. This would enable them be able to read out the users' hashed identifiers and perform a dictionary attack to learn the contents of the users' address book, thereby nullifying the purpose of incorporating SGX into private contact discovery.

C. Subverting Signal's Secure Value Recovery

Signal has also developed a technology called Secure Value Recovery, which is designed to help facilitate secure and private cloud storage [4, 60]. Briefly, they propose a system that allows users to encrypt and decrypt their data on the cloud with their password, while at the same time protecting against offline brute force attacks on the password.

This is accomplished by combining a value expanded from a user's password (called c_1) and a random 256 bit value (called c_2) to derive the data encryption key. The entropy provided by c_2 prevents an attacker from bruteforcing the encryption key, and it is stored within an SGX enclave on the server. In the event that the user loses their encryption key (e.g. loses their phone or latop), the user needs to re-compute c_1 and c_2 to derive the encryption key. The user can present their expanded password to query the enclave for c_2, and can recover c_1 by simply re-expanding her password.

Preventing Brutce Force Attacks. In order to realize the benefit of using c_2 to prevent offline brute force attempts, the SGX enclave must effectively limit the number of times one can query for c_2. Signal accomplishes this by designing an enclave that stores the number of remaining c_2 queries in a Raft [72] distributed log, where the other nodes are also SGX enclaves thay verify each other through remote attestation. This enables the log to live entirely in SGX memory; storing the log on disk is not an option because an attacker can "roll back" the state of the log by simply plugging in an identical hardrive after all attempts have been exhausted.

Leaking c_2. By breaching the confidentiality of the enclave, an attacker can extract c_2 and gain an unlimited number of attempts to brute force users' passwords. The more interesting implication, however, is that if an attacker can compromise any single node used in the Raft consensus protocol, then she can perform the attack from Section IV to gain complete control of that node, and the other nodes will trust that the malicious node is behaving properly due to remote attestation. As noted by Copeland and Zhong [19], Raft's guarantees on consensus dissolve completely in the presence of even a single Byzantine node. The malicious node can trivially become the Raft leader [19]; because the Raft leader is the sole point of contact with the client, the malicious leader can modify the table containing c_2 and the corresponding authentication tokens, and also lie arbitrarily to clients requesting c_2.

VI. Conclusions

In this work we presented SGAxe, a transient execution attack that is able to recover SGX attestation keys from a fully updated Intel machine which is trusted by Intel's attestation server. With access to this key, we demonstrate that we can sign arbitrary attestation quotes, which are subsequently considered genuine by Intel's attestation servers. Thus, SGAxe effectively breaks the most appealing feature of SGX, which is the ability on an enclave to prove its trustworthiness over the network. Finally, our work exposes the fragility of the SGX ecosystem, where a single vulnerability can result in cascading compromises that erode the security and trust properties of SGX.

VII. Acknowledgments

The authors would like to thank Marina Minkin for helpful discussions in the course of this research.

This research was supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under contracts FA8750-19-C-0531 and HR001 120C0087, by the National Science Foundation under grant CNS-1954712, by an Australian Research Council Discovery Early Career Researcher Award (project number DE200101577), and by generous gifts from Intel and AMD.

References

[1] "Enterprise data and cloud security simplified," https://www.anjuna.io/.

[2] "Azure confidential computing," https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[3] "Signal," https://signal.org/en/.

[4] "Introducing Signal PINs," https://signal.org/blog/signal-pins/, 2019.

[5] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *HASP*, vol. 13, 2013.

[6] Arm, "Arm TrustZone technology," https://developer.arm.com/ip-products/security-ip/trustzone.

[7] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi, "Secure multiparty computation from SGX," in *FC*, 2017, pp. 477–497.

[8] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-time cryptocurrency exchange using trusted hardware," in *CCS*, 2019, pp. 1521–1538.

[9] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," in *CCS*, 2019.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *WOOT*, 2017.

[11] S. Brenner, T. Hundt, G. Mazzeo, and R. Kapitza, "Secure cloud micro services using Intel SGX," in *DAIS*, 2017, pp. 177–191.

[12] E. Brickell and J. Li, "Enhanced privacy ID from bilinear pairing for hardware authentication and attestation," *International Journal of Information Privacy, Security and Integrity 2*, vol. 1, no. 1, pp. 3–33, 2011.

[13] A. Cabrera Aldaya, C. Pereida García, L. M. Alvarez Tapia, and B. B. Brumley, "Cache-timing attacks on RSA key generation," *TCHES*, vol. 2019, no. 4, pp. 213–242, 2019.

[14] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on Meltdown-resistant CPUs," in *CCS*, 2019.

[15] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security*, 2019, pp. 249–266.

[16] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races," in *IEEE SP*, 2018, pp. 178–194.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T.-H. Lai, "SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution," in *Euro S&P*, 2019, pp. 142–157.

[18] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *IEEE SP*, 2019.

[19] C. Copeland and H. Zhong, "Tangaroa: a Byzantine fault tolerant Raft."

[20] V. Costan and S. Devadas, "Intel SGX explained," IACR Cryptology ePrint Archive 2016/086, 2016.

[21] F. Dall, G. De Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks," *TCHES*, pp. 171–191, 2018.

[22] Equinix Inc., "Security control for all clouds," https://www.equinix.com/services/edge-services/smartkey/.

[23] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 693–707, 2018.

[24] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: functional encryption using Intel SGX," in *CCS*, 2017, pp. 765–782.

[25] Fortanix Inc., "Fortanix runtime encryption platform," https://www.fortanix.com/assets/Fortanix_RTE_Platform_Whitepaper.pdf, 2019.

[26] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, "SGX-LAPD: Thwarting controlled side channel attacks via enclave verifiable page faults," in *RAID*, 2017, pp. 357–380.

[27] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "HardIDX: Practical and secure index with SGX," in *CODASPY*, 2017, pp. 386–408.

[28] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[29] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *ACNS*, 2018, pp. 83–102.

[30] D. Genkin, R. Poussier, R. Q. Sim, Y. Yarom, and Y. Zhao, "Cache vs. key-dependency: Side channeling an implementation of Pilsung," *TCHES*, vol. 2020, no. 1, pp. 231–255, 2020.

[31] J. Goldbard, M. Marlinspike, and S. Glynn, "MobileCoin: A crypto-currency delivering best user experience in blockchain world," https://static.coinpaprika.com/storage/cdn/whitepapers/4235403.pdf, Nov. 2017.

[32] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *EuroSec*, 2017, pp. 1–6.

[33] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security*, 2015, pp. 897–912.

[34] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[35] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," https://software.intel.com/content/www/us/en/develop/articles/using-innovative-instructions-to-create-trustworthy-software-solutions.html, Aug. 2013.

[36] J. Horn, "Speculative execution, variant 4: Speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[37] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *NDSS*, 2013.

[38] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–32, 2018.

[39] IBM, "IBM cloud data shield," https://www.ibm.com/cloud/data-shield.

[40] *Intel Software Guard Extensions for Linux OS*, Intel, https://github.com/01org/linux-sgx.

[41] Intel, "Deep dive: Intel analysis of L1 terminal fault," https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault, Aug 2018.

[42] ——, "Deep dive: Load value injection," =https://software.intel.com/security-software-

guidance/insights/deep-dive-load-value-injection, Mar 2020.

[43] ——, "Deep dive: Intel analysis of microarchitectural data sampling," https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling, May 2019.

[44] ——, "Deep dive: Intel transactional synchronization extensions (Intel TSX) asynchronous abort," https://software.intel.com/security-software-guidance/insights/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort, Nov 2019.

[45] *Intel SGX and Side-Channels*, Intel, https://software.intel.com/en-us/articles/intel-sgx-and-side-channels.

[46] *Intel Software Guard Extensions*, Intel, https://software.intel.com/sites/default/files/332680-001.pdf.

[47] Intel, "2019.2 IPU – Intel SGX with Intel processor graphics update advisory," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00219.html, Nov. 2019.

[48] *Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation*, Intel, https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf.

[49] *Intel Software Guard Extensions SDK for Linux OS*, Intel, 2016, https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

[50] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES," in *IEEE SP*, 2015.

[51] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "SPOILER: Speculative load hazards boost Rowhammer and cache attacks," in *USENIX Security*, 2019, pp. 621–637.

[52] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel software guard extensions: EPID provisioning and attestation services," White Paper, 2016.

[53] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," *White paper*, 2016.

[54] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[55] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE SP*, 2019.

[56] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *WOOT*, 2018.

[57] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "Pesos: Policy enhanced secure object store," in *EuroSys*, 2018.

[58] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.

[59] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE SP*, 2015.

[60] J. Lund, "Technology preview for secure value recovery," https://signal.org/blog/secure-value-recovery/, 2019.

[61] A. Lutas and D. Lutas, "Security implications of speculatively executing segmentation related instructions on Intel CPUs," https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-INTEL-CPUs.pdf, Aug 2019.

[62] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *CCS*, 2018, pp. 2109–2122.

[63] M. Marlinspike, "Technology preview: Private contact discovery for Signal," https://signal.org/blog/private-contact-discovery/, 2017.

[64] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the other side: SSH over robust cache covert channels in the cloud," in *NDSS*, 2017.

[65] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *HASP@ ISCA*, 2013.

[66] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017, pp. 69–90.

[67] ——, "CacheZoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017, pp. 69–90.

[68] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *USENIX Security*, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-medusa

[69] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *IEEE SP*, 2020.

[70] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone explained: Architectural features and use cases," in *CIC*, 2016, pp. 445–451.

[71] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *USENIX ATC*, 2018, pp. 227–240.

[72] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC*, 2014, pp. 305–319.

[73] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *CT-RSA*, 2006.

[74] C. Percival, "Cache missing for fun and profit," 2005.

[75] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A comprehensive survey," *ACM CSUR*, vol. 51, no. 6, pp. 1–36, 2019.

[76] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *IEEE SP*, 2018, pp. 264–278.

[77] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zero-Trace: Oblivious memory primitives from Intel SGX," in *NDSS*, 2018.

[78] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *IEEE SP*, 2015, pp. 38–54.

[79] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[80] L. Shen, "Fortanix provides Intel SGX-protected KMS with Alibaba cloud," https://www.alibabacloud.com/blog/fortanix-provides-intel%C2%AE-sgx-protected-kms-with-alibaba-cloud_594075, Oct. 2018.

[81] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs." in *NDSS*, 2017.

[82] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," *arXiv preprint arXiv:1806.07480*, 2018.

[83] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[84] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *USENIX ATC*, 2017, p. 8.

[85] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[86] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking transient execution through microarchitectural load value injection," in *IEEE SP*, 2020.

[87] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Rogue in-flight data load," in *IEEE SP*, 2019.

[88] S. van Schaik, A. Milburn, S. Osterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Addendum a to RIDL: Rogue in-flight data load," 2019.

[89] ——, "Addendum 2 to ridl: Rogue in-flight data load," 2020.

[90] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," https://cacheoutattack.com/, 2020.

[91] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *CCS*, 2017, pp. 2421–2434.

[92] S. Weiser, R. Spreitzer, and L. Bodner, "Single trace attack against RSA key generation in Intel SGX SSL," in *AsiaCCS*, 2018, pp. 575–586.

[93] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," https://foreshadowattack.eu/foreshadow-NG.pdf, 2018.

[94] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE SP*, 2015, pp. 640–656.

[95] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," 2020.

[96] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.

[97] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: a timing attack on OpenSSL constant-time RSA," *J. Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[98] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *CCS*, 2016, pp. 270–282.

[99] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *CCS*, 2012, pp. 305–316.

[100] W. Zheng, A. Dave, J. Beekman, R. A. Popa, J. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *NSDI*, Boston, MA, 2017.