SMILE: Secure Memory Introspection for Live Enclave

Lei Zhou*†, Xuhua Ding‡, Fengwei Zhang†*
*Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, 518055, China
†Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, 518055, China
{zhoul6,zhangfw}@sustech.edu.cn
‡School of Computing and Information Systems, Singapore Management University, Singapore
xhding@smu.edu.sg

***Abstract*—SGX enclaves prevent external software from accessing their memory. This feature conflicts with legitimate needs for enclave memory introspection, e.g., runtime stack collection on an enclave under a return-oriented-programming attack. We propose SMILE for enclave owners to acquire live enclave contents with the assistance of a semi-trusted agent installed by the host platform's vendor as a plug-in of the System Management Interrupt handler. SMILE authenticates the enclave under introspection without trusting the kernel nor depending on the SGX attestation facility. SMILE is enclave security preserving as breaking of SMILE does not undermine enclave security. It allows a cloud server to provide the enclave introspection service. We have implemented a SMILE prototype and run various experiments to read enclave code, heap, stack and SSA frames. The total cost for introspecting one page is less than 300 microseconds.**

I. INTRODUCTION

Recent years have seen a booming adoption of Intel SGX among a wide range of applications deployed in a cloud platform, such as machine learning model development [1], [2], access control [3], [4] and secure multi-party computation [5]. In these applications, security-sensitive data and code are placed into an enclave whose hardware based isolation prevents external software, including the kernel/hypervisor, from tampering with its internals. Nonetheless, the security strength of SGX does not fully relieve users from security concerns. An adversary may feed a victim enclave thread with poisonous inputs to exploit its code vulnerabilities leading to control flow hijacking, data leakage, or even code injection. The recent use of enclave libraries such as libOS [6], [7], [8] expands the enclave code and leads to a higher risk of software exploiting.

For non-SGX applications, such threats can be coped with by memory introspection which provides the data needed for attack diagnosis. However, for SGX applications, it is highly challenging to use the same method due to the compound of several factors. First, SGX shields the enclave contents against any software access from the outside. Even if the security requirement is relaxed to trust the kernel, it still cannot read the contents stored in the Enclave Page Cache (EPC) pages. Second, it is imprudent to unconditionally trust the self-report made from the inside. Since there is no hardware-enforced privilege separation within the enclave code, a vulnerability exploit may affect all code. As SGX2 even allows new EPC pages to be added to a running enclave, an exploit can lead to code injection. Third, when the kernel is untrusted, there is no known solution to determine which enclave an EPC page belongs to. SGX local/remote attestation is barely helpful as it is designed for launching time integrity and measures the enclave's virtual memory without asserting physical addresses of EPC pages. Moreover, it is susceptible to side-channel attacks [9], [10], [11], [12], [13] which extricate cryptographic keys from an enclave. For instance, Ragab et al. [14] have demonstrated how to make arbitrary remote attestation after extracting the root provisioning key. Lastly, an introspection result is useful only if it is from the expected running instance of the enclave. If the server's kernel is malicious, it can use the same enclave image and launch it in the adversary's own computer. Replay attacks [15], [16] can divert the introspection request and respond with fake data from the relocated enclave.

In this paper, we overcome the challenges above and present a system called *Secure Memory Introspection for Live Enclave* (or SMILE). A cloud service provider can deploy SMILE-compliant platforms from OEM vendors to provide the enclave introspection service to its clients. SMILE allows the owner of an enclave – and *only* the owner – to retrieve her enclave contents at runtime. Against both enclave software compromise and kernel/hypervisor compromise, it ensures that introspection is upon the expected enclave (**enclave authenticity**) and that the results are not faked by corrupted enclave code (**introspection genuineness**). Moreover, while SMILE security failure compromises enclave authenticity and introspection genuineness, it does *not* undermine the default enclave security. We name this property as **security preserving**. SMILE relies on an agent running in the enclave platform's System Management Mode (SMM). The agent is *semi-trusted* as it is trusted to run SMILE but not trusted to learn enclave secrets, resembling the honest-but-curious server widely used in privacy-preserving outsourced database systems.

The Trusted Computing Base (TCB) of SMILE is the union of the default TCBs for SGX and for SMM [17]. We highlight that the SGX TCB alone is insufficient for memory introspection because SGX neither deals with enclave software compromise nor supports runtime measurement. To

Fengwei Zhang is the corresponding author.

use SMILE, enclave users are expected to trust the enclave hosting platform's OEM vendor (e.g., DELL) which has duly checked the SMM agent's security and ensured its loading time code integrity. We also stress that, thanks to the security preserving property, the trust on the OEM and its authorized SMM code only determines the security of introspection. The trust model on Intel SGX remains unchanged.

We have implemented a prototype of SMILE on a Gigabyte-Q170M motherboard. Depending on the introspection workload used in our experiments, a SMILE based enclave introspection takes a system-wide performance toll varying from about 160 microseconds to a dozen milliseconds, which makes it ill-suited for large scale memory introspection or continuous enclave monitoring. It is a useful tool for scenarios wherein the security or functionality needs dominate performance considerations, e.g., live forensics on enclaves under a software exploitation attack or debugging data collection from enclaves throwing out exceptions. We have also conducted four case studies to show how SMILE is applied to report the enclave's call stack, the SSA region and the code region. Note that the techniques using introspection results for various security purposes are orthogonal to and beyond the scope of our study.

We summarize our key contributions as follows:

- We study the problem of how an enclave owner introspects her live enclave memory with enclave authenticity, data genuineness, and enclave security preserving, and propose SMILE as a solution that leverages a semi-trusted agent running in SMM.
- We innovate the *confined interrogation protocol* which harnesses both system security and cryptography to securely bootstrap trust on an enclave, authenticate its identity, and verify runtime code integrity. The protocol does not use SGX local or remote attestation facility.
- We implement SMILE and and measure its performance. We also run several test cases to show its applications.

ORGANIZATION. The next section briefly explains the background on SMM and SGX. Section III overviews SMILE including the adversary/trust models, problem statement, and our approach. The details of SMILE are presented in Section IV. Section V shows the security evaluation. We describe our prototype implementation in Section VI. Section VII presents the performance overhead, introspection speed, and four case studies of SMILE. We survey the related work in Section VIII. Section IX discusses the limitations. Finally, Section X concludes the paper.

II. BACKGROUND

This section provides a preliminary explanation of Intel SMM and SGX. Experienced readers may skip it.

A. System Management Mode

System Management Mode (SMM) [17] is a highly-privileged CPU execution mode available in all current x86 machines. Its main usage is to handle system-wide functions such as power management and special functions needed by OEM. Only the SMM code with an OEM signature can be accepted and installed by the BIOS/UEFI firmware. It is loaded into a special memory region named System Management RAM (SMRAM). The hardware ensures the SMM code's exclusive access to SMRAM. Namely, no system software can read or write contents in SMRAM. Upon receiving a System Management Interrupt (SMI), the CPU switches from Protected Mode (PM) to SMM and executes the SMI handler. The hardware automatically saves the CPU state in the SMRAM state save area. After handling the SMI, the handler uses `RSM` instruction to exit from SMM to PM so that the interrupted threads continue their executions. The SMI handler's access to the platform's physical memory of the platform is not subject to the kernel's paging access control.

An important property about SMI is that, whenever one core receives it, all cores switch to SMM in a synchronous way. Each core can have its own separate SMI handler as well as a separate SMRAM region to save its own state.

B. Software Guard eXtensions (SGX)

With SGX [18], [19], [20], developers can designate a continuous virtual address region as an SGX enclave. The hardware ensures that no other software including the OS-/hypervisor can read or write the contents inside.

The enclave code and data reside in the Enclave Page Cache (EPC) pages in a region of Processor Reserved Memory (PRM) [21]. Enclave execution can be multithreaded. Like normal application threads, enclave threads share the code section and the heap but have their individual stack. Each thread has its own State Save Area (SSA) which saves the CPU states upon an exception, and its own Thread Control Struct (TCS) which is used exclusively by the hardware to manage thread execution. The enclave code can freely access data outside of the enclave whereas it cannot pass the control to the outside which will trigger an exception.

Enclaves are initialized, launched and managed by the OS. Enclave code's internal memory access still involves the MMU which enforces access controls based on the permission bits saved inside the enclave's metadata during enclave initialization. Nonetheless, the MMU consults the outside page table to determine whether the EPC page to access is present.

III. SYNOPSIS

We consider the following application setting. A cloud service provider has its SGX platform (denoted by H) installed with the SMM code for SMILE (denoted by the SMM *agent*) and provides the security-preserving enclave introspection service to its users. A user (denoted by the *owner*) builds her SMILE-compliant enclave $\mathcal{E}$ and deploys it on H.

A. Models

Adversary Model. The adversary is malware in H with the kernel and the hypervisor privileges. It launches various software attacks, e.g., return-oriented programming attacks [22], [23] and heap buffer overflow [24], by exploiting $\mathcal{E}$'s vulnerabilities. It also aims to defeat the enclave introspection mechanism by using an imposter enclave and/or manipulating

the introspection outcome. It is capable of launching replay attacks [15], [16] by colluding with a remote platform under its full control. In the worst case, the malware possesses a leaked signing key belonging to one of CAs in Intel Attestation Service so that it can make arbitrary attestation. Denial-of-service attacks are out of our scope.

Trust Model. The security of SMILE is built upon the hardware TCB of SGX and the assumed security of the SMM code[1] in H. Since the SMM code cannot be independently verified by a third-party, our trust model for SMILE depends on OEM's liability to govern SMM code installation, an unneeded assumption for trusting SGX. Hence, we suppose that the OEM vendor of H has duly validated and authorized the entire SMM code which encloses the SMM agent as a plug-in. We also suppose that the hardware only loads and launches OEM authorized code in SMM. Moreover, the vendor certifies the agent's signing key, so that an agent's signature is accepted by an enclave owner if a valid certificate is presented.

CAVEAT. Similar to existing SMM-based systems [25], [26], SMILE expands the attack surface against SMM. Nonetheless, SMM compromise does not affect the enclave security. Our design is in line with the trust on honest-but-curious servers in privacy-preserving outsourced databases.

B. Problem Statement

Our research problem is how to securely introspect $\mathcal{E}$'s memory under the aforementioned adversary model. A secure enclave memory introspection is expected to meet the following requirements. (a) The returned bytes are indeed fetched from the desired EPC pages of $\mathcal{E}$ running in H (**authenticity**). They should not be from other enclaves or other instances of $\mathcal{E}$. (b) The introspection result is not tampered with or faked by the corrupted code inside $\mathcal{E}$ (**genuineness**). In our attack model, the code and data in $\mathcal{E}$ running in H may have been altered before introspection. (c) The introspection does not undermine the enclave security. Namely, the adversary cannot take advantage of introspection to break secrecy and integrity of $\mathcal{E}$ (**security-preserving**).

Obviously, authenticity is the prerequisite of a secure introspection and also presents the hardest challenge. Since the enclave authentication cannot use SGX attestation facility, authenticity demands genuine data acquisition from the relevant EPC pages. Hence, we have to break the cyclic dependence between authenticity and introspection. Genuineness is an indispensable mandate for any memory introspection setting, because the security objective of introspection is to cope with potential compromise of the target. The challenge arises from the fact that all enclave code has the same privilege. It is thus infeasible to assume that some enclave code is free from tampering and always runs as expected. The security-preserving requirement is from the usability perspective since enclave introspection should not be realized at the expense of enclave security.

[1]Note that the real-life SMM code in several platforms is found to be exploitable.

C. The Approach

Overview. The high-level workflow of using SMILE is as follows. An enclave owner submits to the cloud her enclave and a public configuration file supplying all information needed for her enclave introspection. The cloud launches the enclave at H and passes the configuration to the SMM agent. At runtime, the introspection proceeds in the following steps.

I. The owner submits an introspection request which specifies her enclave's identity (i.e., `MRENCLAVE`) and the enclave memory addresses and sizes for introspection.
II. The kernel at H passes to the agent the request and the enclave thread's `CR3` which serves as a reference for the agent to locate the enclave.
III. Using the corresponding configuration file, the agent authenticates the referred enclave against the identity in the request, and checks whether the enclave's introspection code is intact. If both affirmed, it signs the request and the configuration and passes the signature with the request to the enclave. The signature is a token proving that the agent has approved the ensuing introspection.
IV. The introspection code in the enclave reads the requested contents and encrypts them together with the agent's signature using the owner's public key. The ciphertext is returned to the owner via an open network channel. The owner accepts the introspection result after verifying the signature with the agent's public key certificate.

As shown in the workflow, the SMM agent's responsibility is to authenticate the enclave and assess the trustworthiness of the introspection code therein. It does not access the enclave's private data, which makes enclave security-preserving feasible. The SMM agent is not enclave specific. It can handle multiple enclaves after receiving their public configurations.

Next, we briefly explain how SMILE achieves authenticity, genuineness, and security preserving. The details and a deeper analysis are presented in subsequent sections.

Enclave Authentication. The security of the workflow above hinges on the security of the agent's authentication against the involved enclave and its introspection code. In SMILE, we introduce the *confined interrogation* protocol wherein the agent sets up the *confined environment* to restrict the enclave's capability (e.g., to restrict the available code page and data page to be used) and challenges it to perform a task only the expected enclave can accomplish.

SMILE requires that $\mathcal{E}$ has two pieces of code participating in the confined interrogation: the *anchor* for trust inception and the *worker* for enclave identity report and memory introspection. The outline of the confined interrogation is as follows. The agent first verifies the anchor's code integrity followed by the worker's, and then verifies the enclave identity reported by the worker. The crux is the first step which, like a trust foothold, lays the security foundation for subsequent steps.

Figure 1 illustrates a system view of the confined interrogation on host H having four CPU cores. Besides the anchor and the worker in $\mathcal{E}$ and the SMM agent, SMILE also comprises the *trampoline* which is launched by the agent to set up

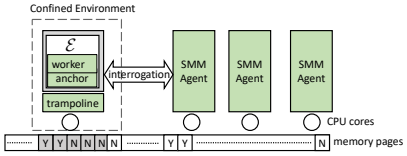


Fig. 1. A system view of confined interrogation. Grey boxes represent EPC pages of $\mathcal{E}$. Only Pages labelled with 'Y' are accessible to $\mathcal{E}$.

the confined environment (details in Section IV-B). During interrogation, $\mathcal{E}$ occupies one core and the agent occupies all others. The confined environment imposes resource restriction on $\mathcal{E}$. It also ensures that the agent directly interrogates the enclave without facing the risk of replay attacks [15], [16] as the enclave is blocked from colluding with other entities including the kernel. Thus, the authentication outcome derived by the agent truly reflects the status of the enclave in engagement.

The confined interrogation protocol (details in Section IV-C) is in the similar vein to software based attestation [27], [28] in the sense that it securely bootstraps the trust on the anchor and then extends the trust chain to the worker. The difference is that our protocol revolves on the space constraint imposed by the confined environment instead of the time constraint [27], [28]. To ensure that only the anchor can pass the checking, we handcraft it with our best expertise so that its memory size is as small as possible. An imposter is forced to occupy more storage and therefore fails to pass the checking due to its conflict with the confined environment. In short, SMILE relies on the interrogation protocol running in tandem with the underlying environment and the anchor to break the aforementioned cyclic dependance.

Genuineness and Security Preserving. Since confined interrogation verifies the code integrity of the introspection code (i.e., part of the worker), genuineness ensues from its confined environment which stalls all untrusted executions. Enclave security is preserved since the SMM agent neither accesses the introspection results nor shares any secret with the owner. All outputs from the enclave during the interrogation are publicly available from the enclave image, and the introspection outcome is encrypted under the owner's public key. Hence, neither a corrupted SMM agent nor an imposter can exploit SMILE to gain any advantage to break $\mathcal{E}$. A comprehensive security analysis is presented in Section V.

IV. Details of SMILE

An enclave introspection transaction using SMILE starts when the kernel in H fires an SMI upon receiving the introspection request from the enclave owner. All the CPU cores are subsequently trapped to SMM, and the SMM agent takes the full control. Next, the agent sets up the confined environment in one of the cores for enclave introspection. We denote it as the *enclave core* for ease of presentation. Then, the agent in one of the remaining cores (denoted as the *interrogation core*) runs the confined interrogation protocol against the enclave in the confined environment. At the end of the protocol, the agent terminates the SMILE transaction by exiting from SMM and returning all cores to the kernel.

In the following, we first describe the enclave internals, before elaborating the confined environment and the interrogation protocol.

A. Enclave Internals

To successfully run in SMILE, the enclave is required to be multithreaded with one thread reserved to launch the anchor. Hence, the developer of $\mathcal{E}$ adds four EPC pages for introspection to the suite of EPC pages for the application logic. The software's view of the enclave layout with access permissions is shown in Figure 2. Specifically, a TCS page and an SSA page are used to launch the introspection thread. Note that SGX denies any software access to TCS pages while it requires the SSA pages to be fully accessible, i.e., with read/write permissions. The new SSA_r page is filled with random bytes chosen by the developer of $\mathcal{E}$. The anchor page and the worker page are *adjacent* which is to facilitate enclave authentication as explained shortly. Note that the virtual address layout of $\mathcal{E}$ is not a secret as it is available in the enclave image.

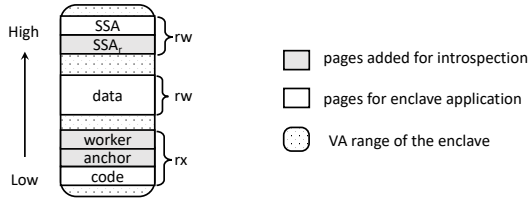


Fig. 2. Software view of the address space layout of an application enclave using SMILE. Only EPC pages accessible to the enclave are shown.

Anchor. The anchor consists of ten instructions only as shown in Figure 3. It begins with copying the SSA page and itself to the non-EPC destination specified by the SMM agent. It then continuously polls a memory location shared with the agent in order to get synchronized with it. It either outputs the worker to a non-EPC destination or jumps to the worker, depending on the bytes in the shared memory. The details of how the segment of instructions run are relevant to the confined environment and the interrogation protocol. We elaborate them in Section IV-B and Section IV-C, respectively.

The anchor code page has two security critical properties. Firstly, the anchor's code size must be as small as possible (i.e., 35 bytes in our implementation). It is within *one* code page and has *no* data page. Secondly, besides the instructions, the page is fully padded with random bytes and no transfer instruction appears in them[2]. Intuitively, the two properties allow the anchor to respond correctly in the confined environment with a memory constraint. More security details are provided in Section V.

Worker. The worker reports the enclave's identity to the SMM agent and makes the due introspection on the enclave.

[2]The developer can scan the random bytes to detect any opcode of transfer instructions. If found, she can either replace it with other bytes or change the operand bytes so that it does not form a legitimate instruction.

```
1:  mov %r10, %rcx                          } Output SSA
2:  lea ssa_offset(%rip),%rsi               }
3:  rep movsd                               }
4:  mov %r10, %rcx                          } Output anchor
5:  lea anchor_offset(%rip), %rsi           }
6:  rep movsd                               }
  loop:                                     } Output Worker
7:  lock xadd %rcx, (%r9)                   } and pass the
8:  jnp worker                              } control to it
9:  rep movsd                               }
10: jmp loop                                }
```

Fig. 3. The anchor code.

Specifically, it first uses the `EREPORT` instruction to get an SGX-generated `REPORT` object whose `MRENCLAVE` member is returned to the agent as the enclave identity. It then makes introspection on the enclave according to the owner's request. The worker page comprises its code and the static data. The code is embedded with the owner's public key, pk, so that the introspection outcome is encrypted using it and safely stored in non-EPC pages before being returned.

B. Confined Environment Initialization and Enclave Entering

After all cores are trapped to SMM, the agent at the enclave core initializes the confined environment and passes the control from SMM all the way down to the enclave execution. Basically, the confinement is imposed by the page tables configured by the SMM agent for the enclave core. Since the agent cannot intervene in runtime events occurring in enclaves (e.g., an exception raised from the enclave), it delegates the runtime checking to the *trampoline* which is loaded by the agent at the enclave core. The trampoline is granted with Ring 0 privilege and intercepts all exceptions and interrupts throughout the interrogation which are abnormal behaviors (e.g., an imposter enclave's attempt to trap to the kernel). The confined environment initialization proceeds in three steps as below.

First, the agent copies the trampoline instructions from SMRAM to the main memory and configures the page table to map it with the supervisor privilege. It modifies the `rip` value in the SMRAM state save area with the trampoline entry address, and exits from SMM. As a result, the trampoline takes the control of the enclave core in Protected Mode.

Next, the trampoline configures the page tables as follows. It sets both the SSA frame and the anchor page as present and all other EPC pages as non-present so that they are not accessible. Since SGX enforces access permissions for EPC pages according to the mappings stored inside the enclave, the trampoline cannot grant or deny permission via the paging tables. However, any reference to a *non-present* EPC page still triggers a page fault. For non-EPC pages, the trampoline maps four pages with priorly prepared contents: one non-writable code page P and three read-writable pages P_0, P_1 and P_2. P only has one instruction which is `EENTER` and is padded with zero for the rest of the space. It provides the inputs for interrogation whereas P_0, P_1 and P_2 are empty pages to store the enclave's interrogation output. No other non-EPC page is mapped for the enclave core. Figure 4 depicts the address space layout with page permissions and presence statuses initialized for the confined environment.

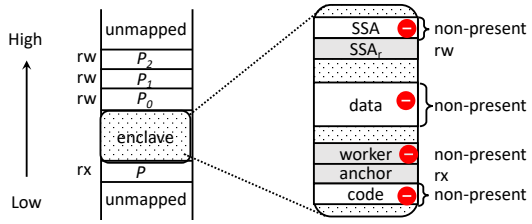


Fig. 4. Initial memory layout for launching the introspection thread of $\mathcal{E}$. Only two EPC pages and four non-EPC pages are made accessible.

Thirdly, the trampoline sets itself as the default exception handler and interrupt handler to intercept any attempted entering to the kernel mode. It prepares the CPU context for the user space execution. Specifically, it sets `rdi` with the address of P_0, `rcx` with 1024, and `r9` with the address of a buffer in P providing interrogation inputs to the enclave. Then, it flushes the TLB to prevent the enclave code from using any cached mappings, which ensures the effectiveness of the page tables prepared for the confine environment. Finally, it launches $\mathcal{E}$ via the `EENTER` instruction in P with the TCS dedicated for introspection.

The whole initialization procedure above is not subject to runtime attacks because, except the enclave core, all other cores are occupied by the SMM agent. Hence, no adversary is live during the procedure. In the end, the enclave core enters $\mathcal{E}$ with no TLB entry storing non-EPC mappings. Since SGX invalidates all TLB entries for EPC mappings upon an enclave exit, this new thread of $\mathcal{E}$ does not use cached EPC mappings either. The enclave thread can only access two EPC pages. The permissions for EPC pages are locked inside the enclave and are in fact unknown to the SMM agent while permissions for non-EPC pages follow the aforementioned setting.

C. Confined Interrogation Protocol

In the confined interrogation protocol, the SMM agent authenticates whether the enclave under interrogation is indeed $\mathcal{E}$. The protocol proceeds in three steps: (i) anchor integrity checking; (ii) worker integrity checking; (iii) enclave identity authentication. Along these steps, the agent progressively builds up trust on the enclave and relaxes the imposed restrictions in the confined environment. The checking against the anchor is the security bedrock because the anchor, after passing the verification, becomes the very first trust foothold in the enclave for subsequent verifications. Consequently, it turns the enclave under interrogation from a black box to a white box.

Anchor Integrity Checking. Since there is no trusted code in the enclave when the interrogation starts, we follow an approach similar to software based attestation [27], [28], [29] to authenticate the anchor. Nonetheless, our mechanism hinges on memory space restriction instead of on execution time as in those schemes. The rationale behind is that *only* the genuine

anchor is small enough to produce the expected result under the stress of memory limitation. Both the anchor page and the SSA frame in $\mathcal{E}$ are fully filled with random bytes except the anchor instructions. The anchor is expected to copy and reproduce *all* random bytes in these two pages to non-EPC P_0 and P_1.

Specifically, when the enclave core enters the enclave for interrogation, the anchor is the first piece of code to run and inherits the CPU context prepared by the trampoline. The relevant registers are shown in Table I. With the memory layout shown in Figure 4, the anchor's first three instructions copy all 4096 bytes in the SSA page to P_0, and then the next three instructions copy all 4096 bytes in the anchor page to P_1. Figure 3 shows the copy instructions.

TABLE I
THE CPU CONTEXT IMMEDIATELY PASSED INTO THE ENCLAVE

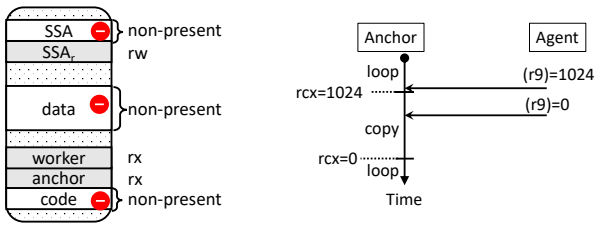
Registers	Content
`rdi`	Address of P_0
`r9`	Address of the shared buffer in P
`r10`	1024
Others including `rsp, rsi`	0

The agent on interrogation core verifies whether P_0 is identical to $\mathcal{E}$'s SSA page and whether P_1 is identical to $\mathcal{E}$'s anchor page. If each and every byte is matched, the agent asserts that the presently running code in the enclave is indeed the expected anchor, though no conclusion can be drawn about the enclave identity.

Note that `rcx` becomes zero after the anchor page is copied. Since the word pointed to by `r9` is also set as zero, the subsequent instructions forms a loop whose payload is equivalent to no operations.

Worker Integrity Checking. After the anchor's code integrity is verified, the agent proceeds to the second phase of the interrogation, i.e., to verify the worker integrity. Specifically, it runs the following steps.

I. Set the present bit in the page table entry corresponding to the worker page, As a result, the worker page is released from restriction. The new memory view is shown in Figure 5(a).
II. Write 1024 to the shared buffer pointed to by `r9`.
III. Repeatedly detect any change on the first byte of P_2. If any, write 0 to the shared buffer.



(a) View of enclave memory with the worker page being released

(b) The interaction between the agent and the anchor

Fig. 5. Worker Integrity Checking

Figure 5(b) depicts the interactions between the agent and the anchor in this phase of checking. The agent's first write to the shared buffer allows the anchor code to load 0x1000 to `rcx`, which leads to the worker page to be copied over to P_2. After the copy operation, `rcx` is reduced to 0 again. By then, the agent has cleared the shared buffer to 0. Hence, the report's loop make no more copy operations.

The agent verifies the outputs in P_2 against the bytes of $\mathcal{E}$'s worker. If all are consistent, it asserts that the enclave under interrogation has the expected worker code.

Enclave Identity Checking. After verifying integrity of the worker, the agent proceeds to authenticate the enclave entity. Specifically, it runs the following steps.

I. Release the EPC data page used for enclave report generation.
II. Write 0xFF to the shared buffer pointed to by `r9`.

As a result, the anchor breaks out of the loop and jumps to the worker (Line 8 of the anchor code in Figure 3). The worker initializes its context including the stack and then executes the `EREPORT` instruction to generate its enclave report. It then returns the `MRENCLAVE` member within the report structure to page P_0.

The agent checks the returned value against $\mathcal{E}$'s known identity. If matched, the agent has successfully authenticated the enclave without using the default SGX local or remote attestation. This completes the confined interrogation protocol.

D. EPC Introspection

After the confined interrogation, the SMM agent prepares for EPC introspection. To avoid straining the agent with the communication task, SMILE is designed to use the normal network channel to submit the introspection result to the owner. The main security problem to overcome is to allow the owner to authenticate whether the received result is indeed made by the worker during a SMILE session. Our solution is to cryptographically bind the SMILE session with the introspection result. Specifically, the agent follows the following steps:

I. Sign the owner's introspection request and the configuration file in use. Let σ be the resulting signature.
II. Pass the request and the signature σ to the worker through the shared buffer.
III. Restore the memory view of $\mathcal{E}$'s process so that the entire enclave is made available.

The worker makes the due introspection and encrypts both the result and σ using the enclave owner's public key pk. It writes the ciphertext blob, $\mathsf{Enc}_{pk}(data, \sigma)$, to the non-EPC page(s) for the application to send it to the owner, permanently deletes σ from the enclave, and notifies the agent the completion of introspection. The agent tears down the confined environment entirely before exiting from SMM to terminate the entire SMILE session.

When the enclave owner receives the ciphertext blob, she decrypts it and verifies σ against her request and the expected configuration file. If both σ and its corresponding pubic key

certificate issued by the OEM are valid, the introspection result is indeed obtained by $\mathcal{E}$ after successfully passing the confined interrogation; otherwise, it is faked.

V. Security Evaluation

In this section, we assess the security of SMILE from two perspectives: how the trustworthiness of the introspection result is achieved via the concerted efforts in system security, algorithm design, and cryptography; and how the enclave security is preserved by SMILE. For the former, we begin with the confined environment which lays the system foundation, followed by analyzing the interrogation protocol. For the latter, we consider various attack scenarios including SMM compromise.

A. Security of Confined Environment

The security objective of the confined environment is to restrict the enclave's memory access as intended and prevent its control flow from leaving the enclave. Throughout the whole interrogation session, only the enclave core runs in PM while all others are in SMM. According to our adversary model in Section III-A, the code and data in SMM is never susceptible to read or modification from PM. Attacks against the confined environment can only take place at the enclave core guarded by the trampoline. Hence, we begin with trampoline security.

Trampoline Security. The trampoline's launching time integrity is guaranteed as it is copied from SMRAM to the main memory. It is not susceptible to runtime software attacks because other cores are controlled by the SMM agent. The malicious kernel may schedule a DMA attack to overwrite the trampoline. There are two approaches to cope with such attacks. If the platform supports IOMMU, the SMM agent configures the IOMMU page tables during trampoline loading to block illicit DMA writes. Otherwise, a randomization based approach can mitigate the risk of attack to certain degree. The SMM agent loads the trampoline to a randomly selected non-EPC page whose contents (if any) are saved before loading and restored after introspection. Depending on the physical memory size, the platform may have hundreds of thousands to millions of page frames for the agent to choose. Since the DMA attack selects the pages to write prior to trampoline loading, this alternative can reduce the attack's success probability, instead of prohibiting it entirely.

User-mode Execution. Throughout the untrusted enclave execution (i.e., the first instruction after entering enclave might be malicious), the trampoline guards at the kernel entry to intercept any user/kernel mode switch. Since the trampoline guards the entire interrogation session, the adversary can only launch attacks in user-mode. Without Ring 0 privilege, it can not alter the page tables, the CPU mode, or tamper with the trampoline.

Memory Access Control. The enclave code can only access non-EPC pages according to the page tables provided by the confined environment, because the trampoline has flushed the TLBs before entering the enclave. Hence, the environment dictates what non-EPC pages are accessible with what permission. For EPC pages belonging to the enclave, the MMU uses the mappings kept inside the enclave. However, a page fault exception is triggered if the present bit in the page table entry is cleared. Hence, the confined environment determines what EPC pages can be accessed though it cannot specify the permissions.

On the premise of the CPU privilege and the memory access control, we examine the enclave's control flow and data flow.

Control Flow. After the interrogation core enters the enclave, any control transfer to instructions outside of the enclave is caught by the trampoline. To leave the enclave, the enclave code must activate either an asynchronous enclave exit (AEX) by triggering events such as a system call or a controlled exit by executing `EEXIT`. The former is trapped to the kernel mode and is intercepted by the trampoline. The latter triggers an exception because no executable page is mapped outside of the enclave. The page fault is also intercepted by the trampoline. Moreover, the in-enclave control transfers are limited within the EPC pages presently mapped, regardless the mappings cached inside the enclave. Specifically, during the anchor integrity checking step, only the anchor page is present while the worker page is also available to execute during the worker integrity checking and the enclave identity checking steps. An attempt to execute code in other EPC pages triggers a page fault.

Data Flow. The enclave code is allowed to make memory references to the non-EPC pages on the condition that the access is compliant to the governing page table entry. Throughout the interrogation session, it can only read P and has the read/write access to P_0, P_1, P_2. No other non-EPC page is accessible to the enclave code. Internally, it has the read/write access to the SSA page. Since the enclave keeps the access permissions to EPC pages, it may have the full access to the anchor page and two pages for the worker.

B. Enclave Authenticity

Since the owner verifies the agent's signature upon the introspection request and the configuration, she can detect request or configuration forgery. Hence, our analysis below assumes that the agent has the correct information about the introspection target, enclave $\mathcal{E}$. Note that the kernel/hypervisor in H must provide all relevant address mappings conforming to $\mathcal{E}$ address layout. Otherwise, exceptions are triggered when a wrong mapping is used. Suppose that the adversary impersonates $\mathcal{E}$ using its own enclave $\mathcal{E}'$ which has the identical address layout as $\mathcal{E}$ but holds different contents. We show below that $\mathcal{E}'$ will be caught by the SMM agent during interrogation.

1) Anchor Authentication: Let A denote the anchor code as shown Figure 3. Suppose that $\mathcal{E}'$ uses its fake anchor A' whose instructions are different from A. Although we do not have a theoretic proof, we reason below that, if A' completes the workflow of interrogation, it occupies more storage than A. Following this, we deduce that it has a negligent chance

of reproducing all random bytes correctly and the spoofing is thus caught by the agent.

During the interrogation, A' must produce the expected 8192 bytes (i.e., anchor page and SSA page), transfer the control to the worker, and synchronize with the agent. To the best of our knowledge, the code in A (Line 1 to Line 6 in Figure 3) is the shortest one copying all bytes in two separated pages in the confined environment. Note $\mathcal{E}'$ cannot place the SSA page and the anchor page adjacent, because it does not match the confined environment setup and will trigger a page fault. We also stress that A' is the very *first* piece of code executed upon enclave entry in the interrogation session and inherits the CPU context (in Table I) prepared by the trampoline. The rationale is that both the CPU context prepared by the SMM agent and A are carefully crafted to fit each other. A' would need extra instructions to initiate its registers using data from the two EPC pages.

The second task is also necessary. The size of A' will be significantly larger if it measures the enclave and performs introspection. It should include one `jmp` instruction to the worker as in Line 8 of A (2 bytes) in Figure 3. Other control transfer instructions such as `ret` and `call` require additional instructions to load `rsp` and the stack, which costs more storage.

The third task is indispensable because A' must run in synchronization with the agent. Otherwise, its behavior will be caught due to the confined environment. It means that A' must continue its execution during the interval between copying two EPC pages out and the worker page restoration. The shortest possible code to run in the confinement environment is a loop that repeatedly polls the shared memory, as used in A which are 8 bytes in total (Line 7 and Line 10 in Figure 3).

Note that during the worker integrity checking step, the agent cannot tell whether the output is made by the anchor or the worker, since A' may have secretly passed the control to the worker. However, this attack requires more bytes than A which completes copying using two bytes (Line 9 in Figure 3). The attack above needs several instructions to fake the genuine instructions outside of its code.

In short, A' uses more storage than A if it passes the anchor checking. However, we show below that there is *no* additional memory space for the adversary to use except the space occupied by A.

A' can only access the SSA frame and its own code page. These two EPC pages are the only storage providing up to 8192 bytes of input data to produce the response for authentication. The non-EPC pages, namely P, P_0, P_1, P_2 do not provide additional information since P_0, P_1, P_2 are empty pages while P only has the `EENTER` instruction. Suppose that A' uses k more bytes than A. It implies that A' has to reproduce $8192 - 35 = 8157$ (size of two pages minus size of anchor) random bytes using $8157 - k$ bytes storage. However, the probability that this 8157-byte long random string can be compressed with k less bytes is $\rho = 2^{-8k+1} - 2^{-8*8157} \approx 2^{1-8k}$, assuming that every random string of such a length has the same probability to be compressible with k less bytes. Hence, the success probability of a meaningful attack (with a large enough k) is overwhelmingly negligible.

To summarize, if the enclave under interrogation returns all bytes as expected, it is confirmed to run the genuine anchor in Figure 3. It means that the anchor becomes the trust foothold for the agent.

2) Worker Authentication & Enclave Authentication: After the anchor is authenticated, its copy of the worker page is also trustworthy. Hence, the agent can securely authenticate the worker's code integrity using $\mathcal{E}$'s image.

Since the worker's enclave identity report does not use any initial data from the enclave and there is no thread running in parallel during its execution, the code integrity directly leads to runtime integrity which means that the outcome of its execution is trustworthy as well. Hence, the agent can use the result from its `EREPORT` to verify the enclave's `MRENCLAVE` for enclave identity authentication.

C. Genuineness of Introspection Result

After the confined interrogation, the SMM agent continues to occupy the CPU cores except the enclave core. Hence, there is no running adversary against the worker. Since the worker's code and data integrity has been verified, its outcome can be trusted by the owner on the condition that it is not susceptible to exploits during introspection, which is a common assumption for all introspection schemes.

Note that the worker does not use any permanent secrets, as it uses the owner's public key which has been enclosed in integrity checking. Hence, even if the adversary invokes the introspection outside of SMILE, it cannot decrypt the outcome. Neither can the adversary impersonate the worker to send a blob to the owner without using SMILE, because it cannot enclose a proper SMM signature σ inside it.

D. Enclave Security Preserving

We explain below that SMILE does not undermine the security of $\mathcal{E}$. Namely, even if the SMM agent is compromised by the adversary, it does not gain additional advantage of attacking $\mathcal{E}$.

First of all, SGX is engineered to resist attacks from SMM. Secondly, the agent neither accesses nor stores any secret pertaining to the enclave as its execution only utilizes data from $\mathcal{E}$'s image. Therefore, the subverted SMM agent does not directly compromise any enclave.

It is more intricate to show that the adversary cannot misuse SMILE to gain advantages. We consider the possible scenario where the adversary runs the interrogation protocol against $\mathcal{E}$ without even trapping to SMM. The anchor is the only code interacting with the adversary. Although the anchor inherits the register context, the adversary can only specify the amount of bytes to copy. It can neither appoint the source (i.e., `rsi`) nor choose an EPC page due to the mask applied to `rdi`. Hence, the anchor will always copies three pages to *non-EPC* pages. Neither of the pages contain sensitive data. The two jump instructions use direct transfer. Thus, the control flow is not susceptible to manipulation. Like other normal enclave

functions, the worker reads parameters from the outside and writes to EPC pages. The use of the worker does not weaken enclave security. In short, the security of SMILE or SMM does not impair the security of SGX. Using SMM as the TCB for SMILE does not expand the TCB of SGX.

E. SMM Agent Security

Under the assumption of a secure SMM agent, we have analyzed how SMILE achieves enclave authenticity, introspection genuineness and enclave security preserving. It is an open problem how for a third-party to assess trustworthiness of the SMM code (including the agent). Since the hardware only accepts the SMM code signed by the vendor, the owner may place her trust upon the vendor's code authorization procedure. Namely, the hardware vendor duly checks the SMM agent to be installed on the cloud provider's platform before signing it and certifying its public key. The enclave owner can assert the SMM agent's loading time integrity if its public key certificate is valid. At runtime, the malicious kernel/hypervisor cannot launch a side-channel attack against the agent's signing key, because all cores are controlled either by the agent itself or the trampoline when the key is used.

VI. PROTOTYPE IMPLEMENTATION

We have implemented a prototype of SMILE consisting the SMM agent, the trampoline module in the kernel, and the anchor thread. Their code sizes are reported in Table II. Our platform is a Gigabyte-Q170M motherboard with a four-core Intel i5-7500 CPU (supporting SGX and SMM). We install Ubuntu 18.04 and use Intel SGX SDK version 2.5 for enclave related coding. Figure 6 shows the implemented components and the high-level workflow.

TABLE II
THE CODE LINES FOR EACH MODULE IN SMILE.

SLOC	SMM Agent	Trampoline	Anchor Thread	
			anchor	worker
Assembly	185	20	10	17
C	-	300	-	<100

A. The SMM Agent

Components. Our SMM agent consists of three components: the *launcher*, the *interrogator*, and the *idler*, running in different cores. The launcher runs on the enclave core to load the trampoline and handle the switches from SMM to Protected Mode (PM). Specifically, it sets the trampoline as the default exception and interrupt handler in kernel, and modifies the CPU context stored in the enclave core's SMRAM state save area, so that the trampoline immediately takes the control when the CPU exits from SMM. The interrogator runs in the interrogation core and implements the confined interrogation protocol as elaborated in Section IV-C. The idler runs in the remaining two cores to hold them in SMM. It is essentially an empty loop which exits after detecting the flag set by the interrogator in the predefined shared memory.

Deployment. The SMM agent is implemented as one part of the SMI handler in SMRAM. One of the challenges is the fact that the entire SMRAM is closed to developers. No user space code or the kernel can access it. We follow the approach in [30] to use Intel DCI-based debugging facility [31] to hack into the SMRAM in our platform. Specifically, we flash a customized BIOS image into the motherboard's firmware. The updated BIOS sets the debugging enabling bits in several JTAG registers such as `HDCIEN` so that the DCI debugger can make read/write accesses to both the host memory and the SMRAM through the JTAG interface.

The debug facility allows us to tackle the second challenge: to integrate our agent with the default SMI handler whose source code is not available. Implementing the agent as an *independent* SMI handler is not a feasible option because it would require the agent to handle all hardware-specific issues for PM-SMM switches as well as CPU initialization. Note that the CPU runs in 16-bit real-address mode upon entering SMM. Hence, the difficulty stems from finding a suitable location to hijack the SMI handling process and run our agent.

Our reverse-engineering shows that each of the four cores has its own separate handler although all handlers have the same logic and a common module. For each of them, we hook our code at the end of initializing the 64-bit real-address mode so that it can access the entire host memory. Specifically, we insert a `jmp` instruction that redirects the flow to the agent located in an unused SMRAM region. Since the hook is before invoking any sub-handler, we use a filter in our agent to resume the original flow if the SMI event is not for SMILE. The destinations of the four inserted `jmp` depend on the expected workload in the cores. For the enclave core, it jumps to the launcher function of the agent; while for the interrogation core, it jumps the interrogator function as shown in Figure 6. For the remaining two cores, it jumps to the idler.

B. The Trampoline Module

Recall that the trampoline is to build and enforce the confined environment for the enclave. It is loaded by the launcher with the kernel privilege so that it can enforce page-level access control and become the first-responder to interrupts and exceptions.

The trampoline runs with the `CR3` passed by the kernel so that it runs in the same hierarchy as the target enclave. The kernel also supplies the VAs of the enclave's TCS and SSA pages. Except the given SSA page, its neighboring anchor page, P, P_0, P_1 and P_2, the trampoline clears the present bit for all other EPC and non-EPC pages in user space. It updates the mappings for P, P_0, P_1 and P_2 to four unused physical pages after zeroing them.

Note that using a false `CR3` or faked mappings for the TCS and SSA only lead to the failure of SMILE. The kernel does not gain any advantage to pass the confined interrogation protocol.

C. Enclave Building

Although the enclave layout required by SMILE (as shown in Figure 2) is fully compatible with SGX's hardware specification, it unfortunately cannot be directly generated by the Intel SGX SDK. Firstly, although the SDK supports

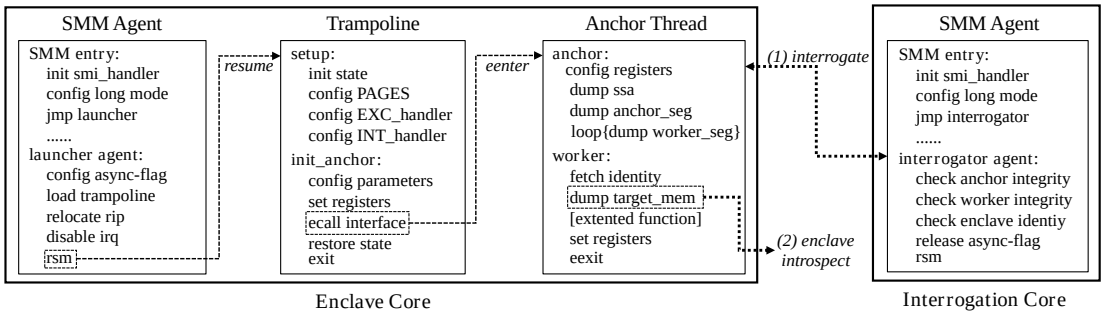
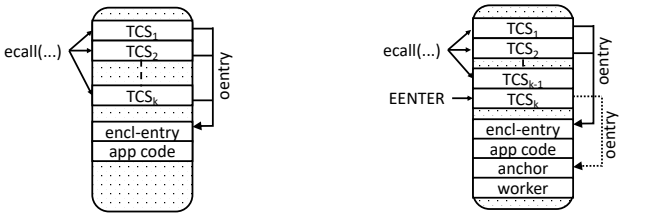


Fig. 6. The implementation details of components and control flow in SMILE.

multithreaded enclaves, all TCS structures' `oentry` fields have the common value as shown in Figure 7(a). Hence, all threads enter the enclave via one entry only. In contrast, SMILE requires the anchor thread to have a dedicated entry which cannot be shared with enclave threads running the application logic. Figure 7(b) shows the desirable TCS setting for SMILE. Secondly, for the sake of security and ease of programming, the SDK injects code in front of the application logic so that the non-enclave code uses the SDK's `ecall` API to invoke enclave functions. However, SMILE requires that the anchor code be the very first to execute after enclave entering in the anchor thread. Even worse, the TCS pages inside the enclave's EPC pages are created by the kernel from the template in the enclave image. Hence, it does not help to add a customized TCS to the image. Neither does work to modify the kernel's enclave creation alone because enclave integrity is broken.

CAVEAT. Note that the adversary is not restricted to use the SDK to build his malicious enclave. Hence, SMILE does not assume the enclave layout prepared by the SDK. On the other hand, since SMILE is to assist enclave developers, it is desirable to be compatible with the SDK for its easy adoption.



(a) Default TCS structures by SDK (b) TCS structures required by SMILE

Fig. 7. Comparison between the default TCS setup and the one for SMILE where TCS_k is different from others.

Hence, we seek a way to overcome these development hurdles with minimal changes on the enclave developer side and the kernel. To use SMILE, the developer still follows the standard way of programming and compiling, but needs to use a modified tool to sign the image. The kernel at the deployment platform needs to use a modified API to create the TCS page required by SMILE. The details are as follows in three steps.

1. Enclave Programming. The enclave developer defines the assembly code of the anchor in Figure 3 as an enclave function. The developer also specifies $k \geq 2$ enclave threads including the anchor thread. The developer compiles her enclave in the usual way and gets the enclave image.

2. Image Signing. By default, the SDK's signing tool first constructs the enclave's memory footprint in non-EPC pages with the same layout and contents as the real enclave to be created in EPC pages, and then produces the signature on the enclave measurement. We modify the signing tool with two changes.

- After it loads the anchor function to the memory, it outputs the anchor's the offset to the enclave base.
- After it creates the last TCS page, it changes the TCS's `oentry` to the anchor's offset to the enclave base so that it will be the entry for a thread using the last TCS.

Hence, when the developer runs the customized tool, she receives the signature upon the enclave layout and contents compliant to SMILE.

3. Enclave Creation. At runtime, the kernel at the deployment platform creates and launches the user enclave. We modify the `build_context()` function in the SGX SDK for the kernel. This function creates an enclave page according to data in the enclave image and load it to the EPC. The modification is similar to the changes upon the signing tool. Namely, it assigns the last TCS's `oentry` to the anchor function entry. These consistent operations allow the created enclave to pass the hardware's integrity checking. We also modify the function that initializes the metadata for the TCS pool to exclude the last TCS reserved for the anchor thread. This modification prevents the `ecall` invocations from the user-space code from entering the enclave via the anchor.

In short, we resolve the incompatibility between SMILE and the Intel SGX SDK by making minor changes on TCS page creation in both the signing tool and the library functions for the kernel. The introduction of the anchor TCS to support SMILE does not affect executions of the enclave's application thread(s).

D. The Worker

Our prototype provides a template of the worker for enclave developers to use. In terms of functionality, the worker template consists of three components: Enclave identity report, memory introspection, and secure output. We use 17 instructions in total to implement the functionality of enclave identity

TABLE III
TIME OVERHEAD OF DEFAULT SMM HANDLING IN SMILE.

	Operations	Time
Default SMM handler	SMM enter & exit	2.1 μs
	Original SMI handling	27 μs

report. It uses the EREPORT instruction with the platform's Quoting Enclave being the target enclave and fetches the MRENCLAVE from the report structure as the enclave identity.

The other two components are C functions. We use the hybrid of 3072-bit RSA and 128-bit AES encryption to encrypt the introspection result with a signature from the SMM agent. Developers can customize them for their own application needs. Note that execution of these two functions are *after* SMILE's confined interrogation protocol. Hence, an improper implementation of them does not affect the authenticity of the concerned enclave. However, we stress that it may leak enclave secrets if the introspection results are not properly encrypted.

VII. EVALUATION

We have run extensive experiments with our SMILE prototype to assess its performance. In the following, we measure the overhead of SMILE as well as the speed of introspection, then describe four cases to demonstrate the usage of SMILE.

A. Overhead of SMILE

We evaluate SMILE's overhead by running a zero-load introspection, i.e., zero byte to fetch from the enclave. By and large, the overhead comprises the time for SMM entering and exit including execution of the default SMI handler (as reported in Table III) and the time for a SMILE session without introspection load.

Since a SMILE session involves two CPU cores, we measure the time interval of major steps on each core. Figure 8 visualizes these intervals with overlaps.

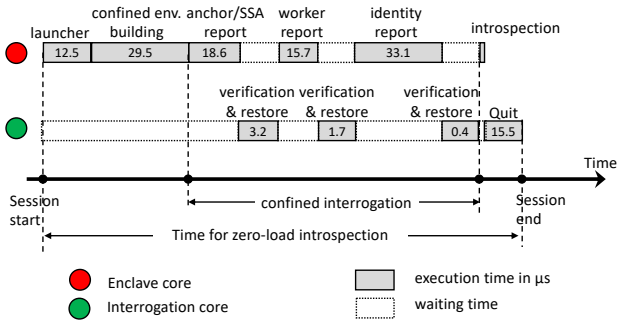


Fig. 8. The execution and waiting time spent in the enclave core and the interrogation core for a zero-load introspection. The sizes of blocks are not proportional to the time interval lengths.

Enclave Core. The launcher agent spends 12.5 μs to load the trampoline and pass the control to it. The trampoline sets up the confined environment with about 29.5 μs, including the overheads for page table configuration, TLB flushing, and enclave entering. After entering the enclave, it costs 18.6 μs and 15.7 μs for the anchor to make two reports: one for the SSA page and itself and the other for the worker page. The worker's enclave identity report takes about 33.1 μs.

Interrogation Core. The interrogator waits until it receives the anchor report. It takes about 8.7 μs to verify it and restore the worker page. The subsequent worker integrity verification costs roughly 1.7 μs to restore the mapping while the enclave identity verification costs 0.4 μs. Lastly, it takes about 15.5 μs to exit from SMM and tear down the confined environment.

Hence, **a SMILE session costs 159.3 microseconds excluding the worker's time for memory introspection**. The cost slightly grows with the enclave size and the application size since more operations are incurred to configure the page table entries. Among the overhead, the interrogation session takes about 72.7 μs, most of which is due to the operations inside the enclave. Note that the time cost of the agent's signature generation is not counted within 159.3 μs because it is in parallel with the worker's introspection including the RSA encryption. Note that the agent can use hash-chained based signature schemes such as SAS [32] whose runtime overhead is merely due to hash operations.

B. Introspection Speed

The worker reads the enclave memory in the same way as other enclave code. Its main overhead is entailed by the hybrid encryption, i.e., one RSA encryption over a randomly generated one-time AES key and an AES encryption over the introspection data. The former takes about 121.7 μs, and the latter's cost is at the rate of 2.1 μs per page. Hence, the total overhead of a SMILE enclave introspection is dominated by the fixed cost of 281 μs including 159.3 μs for interrogation and 121.7 μs for RSA encryption. However, for bulky tasks such as checking the entire code region, the overhead is dominated by the AES encryption on memory pages.

Bulky Introspection. To understand the overhead of a bulky introspection, we run experiments with SGX-FS [33], a secure user-space file system by using Intel SGX. We instrument it with the worker function that reads and encrypts various enclave contents without invoking the confined interrogation. The experiment results are reported in Table IV.

TABLE IV
THE AES MEMORY ENCRYPTION TIME ON SGX-FS.

Section	# of Introspected Pages	Time
Code & Data Sections	172	383.35 μs
Enclave Heap	7,192	14.26 ms
Enclave Stack	4,094	8.21 ms
SSA	4	15.82 μs

As shown in Table IV, when the number of introspected pages is large, the AES encryption alone costs several milliseconds, far above 150 microseconds SMI processing time recommended by Intel [34]. Hence, SMILE is *not* suitable for such introspection workloads. The enclave owner may break a large task into several smaller subtasks so that each subtask's duration is slightly higher than the recommended length. Supposing that a SMILE session only reports r page,

its cost is $281+2.1r$ microseconds. A task for n-page introspection is divided into n/r sessions and costs $281n/r+2.1n$ microseconds in total. However, this simple breakdown faces the consistence issue as the memory under introspection may change due to enclave thread executions between sessions. A more advanced method is that the worker blocks other threads from execution by hooking their common enclave entry in Figure 7(a). Moreover, all sessions can safely use the same AES key without running RSA encryptions for each session. Specifically, the overhead for the first session remains as $281+2.1r$ microseconds and each of the subsequent sessions takes $159.3+2.1r$ microseconds. In total, an n-page introspection task costs $121.7+159.3n/r+2.1n$ microseconds. If setting $r=2$, the full introspection of SGX-FS's heap of 7,192 pages is expected to cost around 588 milliseconds in total, comprising 3,596 sessions with the average session length being 163 microseconds.

C. Applications of SMILE

Due to its performance limitation, we do not recommend SMILE for large scale introspection or continuous enclave monitoring. It is more suitable for situations when the functionality/security demand outweighs performance considerations. For instance, the owner suspects an enclave compromise or the enclave continuously throws out asynchronous exceptions due to bugs. In both scenarios, the owner may use SMILE to collect the enclave's runtime information for forensics or debugging purposes.

We run four case studies to demonstrate potential SMILE applications ranging from the basic problem of enclave code integrity checking to the more intrigue problem of enclave location verification. Due to the length limit, the fourth case is presented in Appendix A.

1) Case 1: Code Integrity Checking: The recently published Plundervolt [35] and VoltJockey [36] are physical attacks that can flip bits in EPC pages. Attackers may use them to modify the enclave code, e.g., changing a branch condition. These modifications cannot be caught by SGX attestation since it reports the enclave's state upon its inception, instead of the runtime state. SMILE can be applied to check the code running in an enclave.

In this case study, we emulate Plundervolt by flipping one bit in the instruction `cmpl $0x1, -0x4(%rbp)` so that it becomes `cmpl $0x0, -0x4(%rbp)`. The attack swaps the conditions for two branches. The comparison between the original bytes and the modified bytes is shown in Figure 9. We use SMILE to copy out all code pages in the enclave. As a verifier, we first compare the hash of the code pages against the expected result. After detecting a mismatch, we further make bit-wise comparison to locate the attacked site.

The security of this SMILE application is similar to the traditional code integrity attestation using a TPM or other forms for root-of-trust for measurement since it only provides a snapshot of the code. It cannot detect ephemeral code changes.

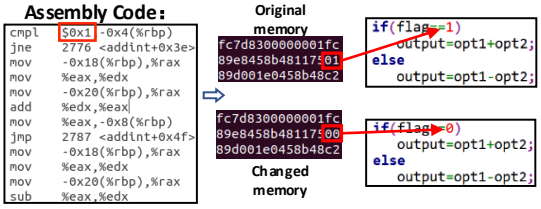


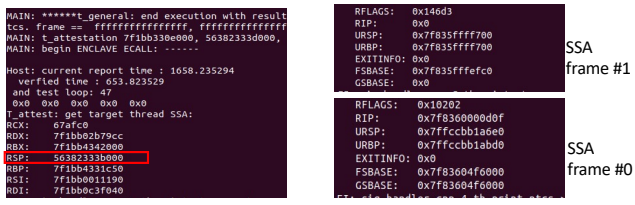
Fig. 9. Memory analysis of enclave code with SMILE.

2) Case 2: SSA State Checking: AEX is a popular attack vector against enclaves. In SGX-Step [37], the adversary uses timer interrupts to nearly single-step the enclave execution for improved side-channel attacks. More powerful attacks are shown in SmashEx [23] where the adversary smashes the enclave stack by interrupting the enclave context restoration during an `ocall` return.

Since the SSA frames of an enclave store the enclave's register state during AEX and are used to resume an interrupted enclave execution, our second case study uses SMILE to dump the entire SSA region for each anchor thread. Note that the SSA contents are not cleared out by the hardware. The results contain rich information for the enclave owner to understand the execution history of the stack. In our experiment, the worker dumps the entire SSA region for each thread. Since all active enclave threads are interrupted by SMI when a SMILE session starts, the forensics function at the owner end begins with the current SSA frame for each thread.

Current SSA Frame. The `GPRSGX` portion of an SSA frame stores the thread's CPU context and other state information. In particular, the owner checks the following:

- whether `RSP` points to the expected address region within the enclave's range. In SmashEx, the attack can change the enclave stack to an non-EPC page fully under the attacker's control. The stack location screening can detect drastic stack relocation attacks. Figure 10(a) shows such a scenario where the enclave stack is outside of the enclave range (`0x7f1bb0000000`,`0x7f1bb8000000`).



(a) Enclave using external stack. (b) Embedded AEX.

Fig. 10. SSA introspection results.

- whether `RIP` points to the expected code region. Note that SGX2 allows the enclave code to modify page permissions. A compromised enclave may collude with the kernel to insert new code.
- whether `uRSP` is in a reasonable range. This field holds the `RSP` used when entering the enclave and will be restored to `RSP` when an AEX occurs. Hence, `uRSP` reflects the outside stack interfacing with the enclave.

Other SSA Frames. SGX organizes an enclave thread's SSA frames in a stack fashion in order to support embedded AEX. Hence, the owner also checks non-empty frames below and above the current one. A frame below indicates that the current SMILE session takes place during an AEX handling. Such a scenario is shown in Figure 10(b) where two non-empty SSA frames are captured. A frame above indicates that there has been embedded AEX events prior to the SMILE session. A large number of non-empty frames in the SSA region is a strong signal indicating external interfering with enclave execution, e.g., the SGX-Step attack. The forensics function further checks the SSA frame's EXITINFO to obtain the exception type.

3) Case 3: Stack Checking: Memory corruption due to enclave code vulnerability is another popular attack vector. SGX-ROP [22] has shown the feasibility of launching an ROP attack inside a victim enclave. SMILE can help a forensics tool to analyze the attack, provided that the adversary does not restore the stack after tampering.

We launch an ROP attack upon our experimental enclave with a stack overflow vulnerability. After the attack, we start a SMILE session introspecting all stack frames. Figure 11 lists the gadget chain in the ROP as well as their manifestation in the runtime stack acquired from the introspection session.

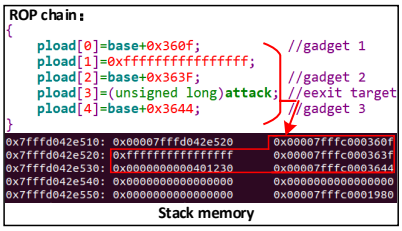


Fig. 11. Memory analysis of stack memory with SMILE.

Note that the worker runs in the anchor thread which uses a different stack from the enclave application thread's. Therefore, the introspection itself does not pollute the attack scene and is not compromised by the gadget chain either.

VIII. Related Work

Our work is motivated by the surge of attacks on enclaves, including passive attacks [9], [14], [38], [39], [40] using a side-channel and proactive attacks [41], [35], [23], [37] by injecting faults. Like other programs, enclave code may have vulnerabilities. Most exploits are code-reuse attacks [22], [42], [43]. However, the growing adoption of SGX2 (which allows EPC permission changes) may meet with more attacks modifying/injecting enclave code modification/inject after an ROP exploit.

Our work is the first that studies enclave live memory introspection. One of its core challenges is to identify the engaged enclave, a security problem related to remote attestation. Intel's SGX remote attestation focuses on proving the initial code and constant data integrity and the legitimacy of the enclave (i.e., properly launched by an Intel certified platform). Although several work such as OpenSGX [44] and OPERA [45] have been proposed to enhance it, these remote attestation schemes neither reflect the runtime enclave code nor check its location. For instance, Biondo *et al.* [43] showed a user-space code reuse attack bypassing attestation. Moreover, the attestation mechanism itself has attracted many side-channel attacks [46] aiming at extracting the secret keys in use. The success of these attacks seriously undermine the trustworthiness of SGX attestation.

Besides the risk of key compromise, the scheme of attestation is susceptible to the replay attack, a problem similar to the one against TPM attestation as described by Parno *et al.* [15]. ProximiTEE [16] detects replay attacks by converting the remote attestation into a distance-bounding protocol with the assistance of an external trusted device. Our scheme does not depend on SGX remote attestation. Our confined interrogation protocol is in the same vein as software-based attestation [27], [28]. Since the confined interrogation protocol does not rely on the response time and CPU speed, it is more reliable than other software-based attestation schemes.

SMILE uses SMM with a different trust model while other SMM-based schemes for various purposes such as integrity checking [47], [26], [48], transparent malware analysis [25], and reliable kernel patching [49]. SMMDumper [50] leverages SMM to directly acquire physical memory, however, it cannot deal with enclave memory. Additionally, the SMM code is *fully* trusted in existing SMM-based systems, while it is semi-trusted in SMILE wherein its compromise does not affect enclave security.

Enclave introspection is a much harder problem than virtual machine introspection [51], [52] where the main obstacle is to cross the boundary set by the target kernel. It is also much harder than normal memory introspection, e.g., SMMDumper [50] which addresses the problem of non-intrusiveness, namely, no data altering due to introspection.

IX. Discussion

SMM Alternatives. SMM is an isolated execution environment that does not depend on the OS running in Protected Mode, so SMILE leverages it to setup the confined environment for enclave memory introspection. Next, we would like to discuss the potential alternatives of SMM. Since SMILE assume that OS is untrusted, one of the alternatives is the Intel Management Engine (IME) [53], an isolated execution environment running within the CPU. Unfortunately, IME might not be a good choice because the communication between IME and CPU is restricted and cannot meet the needs of SMILE without hardware modification. Another alternative is the hypervisor because it can intercept the above OS and provide a confined environment for SMILE; note that the TCB of hypervisor is larger compared to SMM.

SMM and SGX Security. SMM and SGX enclaves are the two key components involved in SMILE, so we might wonder which one is more secure? On one hand, SMM is not designed for security while SGX aims to provide a secure enclave that excludes SMM out of its TCB, so SGX seems to provide a

better security. On the other hand, we believe SGX is more complex than SMM, since enclave is intended for executing user-level applications while SMM is only for supporting several system functions; moreover, the attack surface of SMM may be smaller than that of SGX, because the number of interactions (i.e., SMI/`rsm` or `ecall`/`ocall`) with untrusted code is less for SMM. Both SGX and SMM are potentially vulnerable to side-channels [9], [14], [54], [55] and software bugs [56], [57], however, we believe the security levels of SMM and SGX are not directly comparable, and SMILE leverages them to enable enclave introspection at runtime.

Though SMM is in the TCB of SMILE, it does not weaken the security of SGX. As shown in the security analysis of Section V, a compromised SMM does not break SGX guarantees. SMILE needs to add a small amount of logical code in SGX SDK, and enter the enclave anchor thread without using the SDK provided interface. This bypasses the checking function in the enclave. However, the anchor thread is protected by our confined execution environment, which does not degrade the security.

X. Conclusion

This paper presents SMILE, a novel system for SGX enclave live memory introspection with enclave authenticity, data genuineness, and security-preserving assurance. At the core of SMILE is the confined interrogation protocol that harnesses the power of x86 SMM and cryptography to securely bootstrap trust on an enclave. We have implemented a prototype of SMILE on COTS hardware. The overhead of SMILE based introspection comprises a fixed overhead of 281 microseconds and the in-enclave encryption overhead growing linearly with the workload. While SMILE is not fit for bulky introspection or continuous enclave monitoring, it is the first tool that empowers an enclave owner to collect on-demand runtime data from her enclave under a software exploitation attack.

Acknowledgments

We thank the anonymous shepherd and reviewers for their valuable comments. This work is partly supported by the National Natural Science Foundation of China under Grant No. 62002151, and Science, Technology and Innovation Commission of Shenzhen Municipality under Grant No. SGDX20201103095408029.

References

[1] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-party Machine Learning on Trusted Processors," in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 619–636.

[2] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "SGX-bigmatrix: A practical Encrypted Data Analytic Framework with Trusted Processors," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1211–1228.

[3] S. Contiu, R. Pires, S. Vaucher, M. Pasin, P. Felber, and L. Réveillère, "IBBE-SGX: Cryptographic Group Access Control Using Trusted Execution Environments," in *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 207–218.

[4] J.-B. Djoko-Takougue, J. R. Lange, and A. J. Lee, "NeXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-side SGX," in *Proceedings of the 49th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 401–413.

[5] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi, "Secure Multiparty Computation from SGX," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, 2017, pp. 477–497.

[6] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017, pp. 645–658.

[7] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A Commodity Obfuscation Engine on Intel SGX," in *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[8] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and Efficient Multitasking inside a Single Enclave of Intel SGX," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 955–970.

[9] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX Fails in Practice," https://sgaxeattack.com/, 2020.

[10] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking Data on Intel CPUs via Cache Evictions," *arXiv preprint arXiv:2006.13353*, 2020.

[11] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.

[12] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 557–574.

[13] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 973–990.

[14] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, 2021.

[15] B. Parno, J. M. McCune, and A. Perrig, *Bootstrapping Trust in Modern Computers*. Springer Science & Business Media, 2011.

[16] A. Dhar, I. Puddu, K. Kostiainen, and S. Capkun, "ProximiTEE: Hardened SGX Attestation by Proximity Verification," in *Proceedings of the 10th ACM Conference on Data and Application Security and Privacy*, 2020, pp. 5–16.

[17] Intel, "64 and IA-32 Architectures Software Developer's Manual," https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, 2021.

[18] F. Mckeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[19] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[20] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[21] V. Costan and S. Devadas, "Intel SGX Explained." *IACR Cryptology ePrint Archive*, 2016.

[22] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[23] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, "SmashEx: Smashing SGX Enclaves Using Exceptions," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, p. 779–793.

[24] S. Zonouz, M. Zhang, P. Sun, L. Garcia, and X. Liu, "Dynamic Memory Protection via Intel SGX-supported Heap Allocation," in *Proceedings of the 2018 International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, 4th International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress*, 2018, pp. 608–617.

[25] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using Hardware Features for Increased Debugging Transparency," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, May 2015.
[26] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010, pp. 38–49.
[27] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust." in *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2012, pp. 1–15.
[28] V. D. Gligor and S. L. M. Woo, "Establishing Software Root of Trust Unconditionally." in *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2019.
[29] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "A Security Framework for the Analysis and Design of Software Attestation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 1–12.
[30] M. Ermolov and M. Goryachy, "Intel VISA: Through the Rabbit Hole," *Black Hat Asia*, 2019.
[31] M. Goryachy and M. Ermolov, "Where There's a JTAG, There's a Way: Obtaining Full System Access via USB," https://www.ptsecurity.com/ww-en/analytics/where-theres-a-jtag-theres-a-way, 2019.
[32] X. Ding, D. Mozzacchi, and G. Tsudik, "Experimenting with server-aided signatures," in *Proceedings of the 2002 Network and Distributed System Security Symposium (NDSS)*, 2002.
[33] D. Burihabwa, P. Felber, H. Mercier, and V. Schiavoni, "SGX-FS," https://github.com/dburihabwa/sgx-fs, 2018.
[34] Intel, "BIOS Implementation Test Suite," https://biosbits.org/news/bits-365/, 2011.
[35] 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 *Proceedings of 41st IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 1466–1482.
[36] P. Qiu, D. Wang, Y. Lyu, R. Tian, C. Wang, and G. Qu, "VoltJockey: A New Dynamic Voltage Scaling based Fault Injection Attack on Intel SGX," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.
[37] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *Proceedings of the Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.
[38] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2017, pp. 69–90.
[39] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2017.
[40] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," in *Proceedings of the 2018 Usenix Annual Technical Conference*, 2018, pp. 227–240.
[41] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, "VoltPillager: Hardware-based Fault Injection Attacks against Intel SGX Enclaves using the SVID Voltage Scaling Interface," in *Proceedings of the 30th USENIX Security Symposium*, 2021.
[42] M. Schwarz, S. Weiser, and D. Gruss, "Practical Enclave Malware with Intel SGX," in *Proceedings of the 2013 International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019, pp. 177–196.
[43] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 1213–1227.
[44] P. Jain, S. J. Desai, M.-W. Shih, T. Kim, S. M. Kim, J.-H. Lee, C. Choi, Y. Shin, B. B. Kang, and D. Han, "OpenSGX: An Open Platform for SGX Research," in *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, 2016, pp. 21–24.
[45] G. Chen, Y. Zhang, and T.-H. Lai, "OPERA: Open Remote Attestation for Intel's Secure Enclaves," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2317–2331.
[46] 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," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018.
[47] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "HyperCheck: A Hardware-assisted Integrity Monitor," in *IEEE Transactions on Dependable and Secure Computing*, 2014, pp. 332–344.
[48] J. Rutkowska and R. Wojtczuk, "Preventing and Detecting Xen Hypervisor Subversions," http://www.invisiblethingslab.com/resources/bh08/part2-full.pdf, 2008.
[49] L. Zhou, F. Zhang, J. Liao, Z. Ning, J. Xiao, K. Leach, W. Weimer, and G. Wang, "KShot: Live Kernel Patching with SMM and SGX," in *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 1–13.
[50] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi, "When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition," in *Proceedings of the 2012 Annual Computer Security Applications Conference*, 2012.
[51] Y. Fu and Z. Lin, "Space Traveling Across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy(S&P)*, 2012, pp. 586–600.
[52] S. Zhao, X. Ding, W. Xu, and D. Gu, "Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed," in *Proceedings of the 26th USENIX Security Symposium*, 2017.
[53] L. Zhou, J. Xiao, K. Leach, W. Weimer, F. Zhang, and G. Wang, "Nighthawk: Transparent system introspection from ring-3," in *Proceedings of the European Symposium on Research in Computer Security*, 2019, pp. 217–238.
[54] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow," *Proceedings of the IEEE Micro*, pp. 66–74, 2019.
[55] 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 *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 142–157.
[56] L. Duflot, O. Levillain, B. Morin, and O. Grumelard, "Getting into the SMRAM: SMM Reloaded," in *Proceedings of the 12nd CanSecWest Conference*, 2009.
[57] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, Mickey Shkatov, "A New Class of Vulnerabilities in SMI Handlers," http://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf, 2015.

APPENDIX

A. Case 4: Enclave Location Verification

In a typical SGX application, an enclave owner remotely communicates with her enclave running a designated server. After the owner verifies the enclave's remote attestation, a secret key is shared between them to establish a secret network channel. Due to security and legal needs, the owner may demand to ensure whether the running enclave (i.e., the endpoint of the secure channel) is indeed located on the designated server. However, as noted in [16], Intel SGX's remote attestation facility does *not* assert the physical location of the enclave due to the privacy protection. Figure 12 depicts the attack scenario exploiting the limitation of SGX attestation. The owner of $\mathcal{E}$ expects it to run on Host H. However, the malicious kernel in H collude with the adversary physically controls H'. The relay attack in [16] allows $\mathcal{E}$ in H' to pass the attestation checking and subsequently establish the secure channel with the owner.

SMILE offers a simple solution to verify the location of the enclave sharing the secret key with the owner, supposing that the owner trusts the SMM agent in H to honestly run SMILE. As shown in Figure 12, the owner initiates a SMILE session to introspect the shared secret key used in the present secure

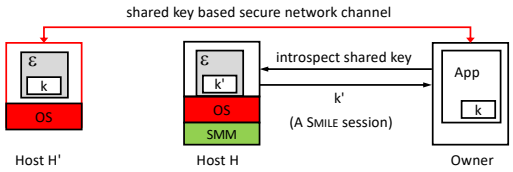


Fig. 12. The adversary controls the OS in H and the entire H'. The owner interacts with her enclave in H' instead of H via a secure network channel.

channel. If the outcome is identical to her current key for the channel, she is ensured that she is communicating with $\mathcal{E}$ presently running in H, because only the SMM agent in H has the signing key to generate the signature enclosed in the introspection result. In our experiments, we set the secret shared key derived from the remote attestation as a global variable so that the worker directly accesses this variable.

REMARK. The recent Data Center Attestation Primitives (DCAP) released by Intel for cloud service providers can bind a provisioning key with an identifiable processor identity by using certificates. However, it is not an ideal solution for individual users. Moreover, the trust on DCAP implies unconditional trust on those CAs managed by Intel, which may conflict with some organisation's policy which distrusts computers beyond their administration. SMILE offers a nimble and lightweight alternative to DCAP.