# Defeating Speculative-Execution Attacks on SGX with HYPERRACE

Guoxing Chen
The Ohio State University
chen.4329@osu.edu

Mengyuan Li
The Ohio State University
li.7533@osu.edu

Fengwei Zhang
Southern University of Science and Technology
zhangfw@sustech.edu.cn

Yinqian Zhang
The Ohio State University
yinqian@cse.ohio-state.edu

***Abstract*—Speculative-execution attacks, such as SgxSpectre, Foreshadow, and MDS attacks, leverage recently disclosed CPU hardware vulnerabilities and micro-architectural side channels to breach the confidentiality and integrity of Intel Software Guard eXtensions (SGX). Unlike traditional micro-architectural side-channel attacks, speculative-execution attacks extract any data in the enclave memory, which makes them very challenging to defeat purely from the software. However, to date, Intel has not completely mitigated the threats of speculative-execution attacks from the hardware. Hence, future attack variants may emerge.**

**This paper proposes a software-based solution to speculative-execution attacks, even with the strong assumption that confidentiality of enclave memory is compromised. Our solution extends an existing work called HyperRace, which is a compiler-assisted tool for detecting Hyper-Threading based side-channel attacks against SGX enclaves, to thwart speculative-execution attacks from within SGX enclaves. It requires supports from the untrusted operating system, e.g., for temporarily disabling interrupts, but verifies the OS's behaviors. Additional microcode upgrades are required from Intel to secure the attestation flow.**

***Index Terms*—Intel SGX, speculative-execution attacks, remote attestation**

## I. INTRODUCTION

Intel Software Guard eXtensions (SGX) is introduced in recent Intel processors aiming to protect data and code within a secure *enclave* against its untrusted host operation systems (OS) or even rogue system administrators. Due to its promise of shielded execution, both researchers and practitioners have built various software tools and applications with these features, *e.g.*, [1]–[11].

The recently disclosed speculative-execution attacks, *i.e.*, Meltdown and Spectre attacks enable a malicious program to read memory content outside its security domain (*e.g.*, reading kernel data from userspace). Their variants, SgxPectre [12] and Foreshadow [13], specifically target Intel SGX to read enclave memory content, completely breaking the confidentiality guarantee of SGX. The newly disclosed Microarchitectural Data Sampling (MDS) hardware vulnerabilities [14]–[16] could also enable the adversary to read enclave memory on-the-fly. Though microcode patches have been released to mitigate these attacks, the fix does not remove the root cause of the vulnerabilities—speculative executed instructions beyond security boundary check and legitimate control flows. Therefore, new variants of such attacks may be discovered in the future.

In this paper, we aim to address speculative-execution attacks against Intel SGX. Existing solutions typically proposes to either prevent speculative execution of suspicious memory loads or close cache side channels [17]–[19], but our solution is software-based. It (1) requests the untrusted operating system (OS) to create a special execution condition for enclaves, under which speculative-execution attacks are impossible; (2) verifies such execution condition is met dynamically at runtime and provides a proof through remote attestation; (3) utilizes an extended SGX feature that is implementable in microcode to guard the attestation keys from memory leaks.

The base of our solution is HYPERRACE, which is a software framework designed to eliminate all micro-architectural side-channel threats due to Hyper-Threading and Asynchronous Enclave eXit (AEX) [20]. Particularly, it creates an auxiliary enclave thread to occupy the sibling hyper thread co-located on the same physical core. Since thread scheduling is performed by the OS, which is untrusted, HYPERRACE runs a statistical tests to verify the threads' co-location.

To guard against speculative-execution attacks that leak any data in the enclave memory, HYPERRACE itself is not enough. In this paper, we propose to disallow interrupts during the invocation of enclave `ECall` functions such that secrets are only allowed to be unsealed into the memory during invocations. HYPERRACE is used to detect the concurrent use of hyper threads so that speculative-execution attacks that leak data from Hyper-Threading is prevented. Eliminating interrupts during `ECall` function invocations require modification of OS kernel to suppress interrupts momentarily; the enclave code needs to verify the occurrence of interrupts at the end of it using techniques proposed by Cloak [21], which is also used in HYPERRACE to detect AEX. As such, no speculative-execution attacks is possible under such an execution condition.

We performed a security analysis of the proposed solution, implemented the kernel components and enclave modules that create the interrupt-free execution windows, and evaluated its performance. The main *contribution* of this work is that it is the first attempt to mitigate the threats of speculative-execution attacks from software, even assuming the possibility of new speculative-execution attack variants discovered in the future. This idea follows with Intel's philosophy of TCB recovery [22] that aims to minimize the security risks enclave data in the

events of SGX compromises.

## II. Background and Related Work

### A. Intel SGX

Intel Software Guard eXtensions (SGX) is a new hardware feature introduced on recent Intel processors, aiming to improve the security of application code and data. Sensitive information can be processed with a shielded execution environment called *enclave*, where the code and data are stored in Processor Reserved Memory (PRM), a dedicated region of the DRAM. Any access to an enclave's memory within the PRM from any software outside of the enclave, even from the privileged softwares, will be denied.

**Asynchronous Enclave eXit (AEX).** To prevent sensitive information from leaking when interrupts or exceptions occur during the enclave's execution, an event called Asynchronous Enclave eXit (AEX) will be triggered. Particularly, before transferring control to the OS, the processor saves the enclave's execution state in a specific enclave memory area called State Save Area (SSA). When resuming the enclave's execution after the interrupts or exceptions are handled, the processor restores the enclave's execution state from SSA.

**Remote Attestation.** To establish trust between enclaves and their remote users or clients, remote attestation is introduced to prove to the remote party that the enclave is running inside an SGX enabled platform. In current SGX remote attestation design, an anonymous signature scheme, called Intel Enhanced Privacy ID (EPID), is adopted [23]. Particularly, an EPID private key is provisioned by Intel Provisioning Service to each SGX platform for generating attestation signatures, which could be verified by Intel Attestation Service. A privileged enclave, called Quoting Enclave (QE) is issued by Intel to manage the attestation key and sign attestation data.

**Root of Trust.** A *root provisioning key* is generated at Intel's internal key generation facility, and burnt into the e-fuse of SGX platforms. The root provisioning key, as a shared secret, establishes the root of trust between Intel and the underlying SGX platform. Besides the root provisioning key, another *root seal key*, which is not known by Intel, is also burnt into the SGX processor during the manufacturing [24]. As root secrets, these two keys are never exposed in the memory but accessed by the `EGETKEY` instruction to derive other keys, which will be exported to enclave memory for various purposes such as encryption and decryption.

**Local Attestation.** Local attestation is used for one enclave to attest itself to another enclave on the same SGX platform. For example, to request Intel's Quoting Enclave to generate remote attestation signature, an attesting enclave needs to use local attestation to prove itself to Intel's Quoting Enclave. Particularly, local attestation is achieved via two SGX instructions: `EREPORT` and `EGETKEY`. `EREPORT` is called by the attesting enclave to derive the report key of the target enclave and generate a cipher-based message authentication code (CMAC) of the report data. Upon receiving the report data, the target enclave executes the `EGETKEY` instruction to derive its report key to generate a CMAC of the report data and compare it with the one in the report. If they match, the verification passes. The derivation of report keys uses both the root provisioning key and the root seal key, so that the local attestation report could only be verified locally on the same platform.

**Sealing.** Sealing is a process used by enclaves to encrypt and integrity-protect some secrets to be stored outside the enclave. The encryption is performed using a seal key, which is derived via the `EGETKEY` instruction. Both the root provisioning key and the root seal key will be used in the derivation. Additionally, a 64-bit `KeyID` can be specified during the derivation, enabling an enclave to derive a large number of different seal keys. The derivation of seals keys uses both the root provisioning key and the root seal key, so that the sealed data could only be unsealed locally on the same platform.

**Trusted Platform Services.** Intel SGX provides trusted platform services, *i.e.*, trusted time and monotonic counters. These services are enabled by the Intel Converged Security and Management Engine (CSME). An Intel issued privileged enclave, called Platform Service Enclave (PSE) is responsible for communicating with the CSME and supports these services.

**Intel Hyper-Threading.** Intel Hyper-Threading is Intel's proprietary implementation of simultaneous multithreading (SMT). When Hyper-Threading is enabled, a single physical core could execute two separate code streams (called hyper threads) concurrently. These two hyper threads, regardless of running in the SGX enclave mode or not, share various resources on the same physical core, *e.g.*, line fill buffers, store buffers, branch prediction units (BPU) and translation lookaside buffers (TLB).

### B. Speculative-Execution Attacks and Defenses

Modern CPUs use speculative execution techniques to increase instruction-level parallelism and hence improve CPU performance. Instructions may be executed speculatively with regard to their program order but are required to retire in order. Security checks take place concurrently to the execution of the instructions, allowing some speculatively executed instructions to access data across the defined security boundary. Inaccurate speculation are discarded at retirement.

**Speculative-execution attacks.** Performance optimization features in modern CPUs have been exploited by various speculative-execution attacks. These attacks execute instructions speculatively to bypass security checks to access secrets in another domain, *e.g.*, accessing kernel memory or enclave memory, and exfiltrate secrets using micro-architectural side channels [25], [26]. According to Canella *et al.* [27], these attacks can be categorized into Meltdown-type attacks [13]–[16], [28], [29] and Spectre-type attacks [12], [30]–[34].

Close to this work are speculative-execution attacks that work on Intel SGX, such as SgxPectre [12], Foreshadow [13], Micro-architectural Data Sampling [14], [16]. These attacks have demonstrated successful extraction of Intel's secrets from

unpatched SGX processors, breaching the confidentiality and integrity of SGX enclaves.

- SgxPectre abuses the branch prediction unit (BPU) to mislead the victim enclave to speculatively execute some secret-leaking gadgets [12]. One key requirement is for the adversary to pollute the BPU which is shared by the two hyper threads on the same physical core, by running malicious code on the sibling hyper thread of the enclave thread, or on the same hyper thread and interrupt the enclave's execution to pollute the BPU .
- Foreshadow targets the enclave secrets that reside in the L1 cache [13]. Hence, the attack could be launched on the sibling hyper thread to leak secrets on-the-fly. Furthermore, even when the enclave is not running, a privileged adversary could abuse `EWB` (SGX instruction for evicting an enclave page, *i.e.*, EPC page, to main memory) and `ELD` (SGX instruction for loading an EPC Page back) to load the enclave data to the L1 cache to launch attacks.
- MDS attacks leak in-flight secret data from CPU-internal buffers (*e.g.*, line fill buffers, and store buffers) [14]–[16]. Similar to Foreshadow attacks, MDS adversaries have to launch attacks on the sibling hyper thread, or collect secrets from the same hyper thread by repeatedly preempting the enclave's execution.

**Existing defenses.** Hardware-based solutions to speculative-execution attacks introduce new hardware features to prevent data leakage [17]–[19]. Specifically, Invisispec [17] removes observable side-effects from data caches; Dynamically Allocated Way Guard (DAWG) [18] isolates caches into protection domains to prevent data leakage from cache side channels; SafeSpec [19] introduces shadow structures to store speculative instructions so that speculatively executed instructions are invisible to the attackers. However, these defenses are yet to be adopted in commodity processors. Practical mitigation to speculative-execution attacks on SGX heavily relies on microcode patches from Intel, which are likely to be bypassed again once new attack variants are discovered.

Software-based solutions are less explored. Gulmezoglu *et al.* proposed *FortuneTeller*, which leverages deep learning techniques to predict microarchitectural attacks, including speculative-execution attacks [35]. Particularly, FortuneTeller collects program's execution traces using hardware performance counters for training and prediction. However, since hardware performance counters are not available within SGX enclaves, FortuneTeller could not be used for detecting speculative-execution attacks against SGX enclaves. In contrast, our work is designed for Intel SGX. And our design is free of false negatives while deep learning based scheme is inevitable of false negatives even if hardware performance counters are extended to be accessible within the enclave.

### C. HyperRace

HyperRace is proposed by Chen *et al.*, aiming to close all Hyper-Threading side channels [20]. The idea is that when enclave code is running on one hyper thread, an auxiliary enclave thread, called shadow thread, will be scheduled by the OS on the sibling hyper thread so that no malicious program could share the same physical core with the enclave program. Since the OS is untrusted, a co-location test is proposed to verify that the shadow thread does run on the sibling hyper thread. Specifically, the co-location tests are based on the contrived data races when the two enclave threads access a shared variable. Particularly, the two enclave threads keeps writing their own chosen unique values to the shared variable and then read from it. When one thread reads a value different from that she wrote, a data race is observed. Leveraging the fact that two hyper threads on the same physical core share L1 cache and two hyper threads on different physical core communicates via last level cache (LLC), the authors designed the data race scheme such that when the two hyper threads are co-located on the same physical core, the measured data race probabilities will be close to 1, and 0 otherwise. Hypothesis testing is used to increase the confidence of the test results. HyperRace also adopted AEX detection technique introduced in Cloak [21] to save co-location tests when no AEX is detected after a successful co-location test. It leverages the fact that AEX will overwrite the SSA. By setting a marker in the SSA, the occurrence of an AEX can be detected be check whether the marker is changed. To date, HyperRace remains the best known approach to detecting Hyper-Threading-based and AEX-based side-channel attacks against SGX.

## III. Threat Model

This paper focuses on speculative-execution attacks against SGX enclaves from a malicious OS. Although known attacks have been mostly addressed by microcode updates, we expect new attacks may at some point breach the confidentiality of the enclave memory again. Therefore, in the context of this paper, we assume (1) the confidentiality of the enclave memory is broken under certain conditions; and (2) the untrusted OS launches, suspends, resumes, and terminates SGX enclaves at will. Without the confidentiality of the attestation keys, however, integrity of the enclaves cannot be maintained.

We assume the processor manufacture, Intel, is trusted, and the processors' internal storage is secure. About the adversary's capability of performing speculative-execution attacks, we assume one of the follow conditions must be met:

- (**R1**) **Attack via concurrent execution**: When Hyper-Threading is enabled, the malicious code can execute on the same physical core concurrently as the victim enclave.
- (**R2**) **Attack via `AEX`**: The malicious OS needs to explicitly preempt the execution of the enclave via `AEX`, by directly or indirectly generating interrupts[1] or triggering exceptions.
- (**R3**) **Attack after `EEXIT`**: Some attacks could be launched even when the enclave code is not running. For instance, Foreshadow could load data into the L1 cache by invoking `EWB` and `ELD` and then perform attacks without triggering the enclave code.

[1]Executing `EWB` will indirectly interrupt all threads of the enclave to flush invalidated TLB entries.

**Algorithm 1:** Protected `ECall` function

**Input:** Inputs
**Output:** Outputs, report

```
1 Set marker in SSA;
2 Co_Location_Test();
3 Outputs ← ECall_Func(Inputs);
4 if the marker in SSA is not changed then
5 |   report ← Gen_Report(Outputs);
6 else
7 |   report ← NULL;
8 return (Outputs, report)
```

## IV. Design

In this section, we describe our design for defeating speculative-execution attacks by extending HyperRace. Our design consists of three parts. The first is to detect speculative-execution attacks within one enclave. The second is to enhance existing attestation loop to prove to the remote client that the attesting enclave is free of such attacks. The third is to secure the trusted platform services to prevent speculative-execution attacks via replay.

### A. Attack Detection

An enclave program usually provides one or more `ECall` functions to fulfill sensitive jobs. Each job can be done by one or multiple invocations of these `ECall` functions. The adversary might launch speculative-execution attacks during one invocation or between two invocations. We first describe how to detect speculative-execution attacks within one invocation, and then introduce the mechanism to protect secrets through multiple invocations.

**Detection within one invocation.** Recall the attack requirements in Sec. III, HyperRace naturally addresses **R1**. But it has limited ability to address **R2**, because HyperRace's detection policy has to allow small amount of blocks to be interrupted for free to tolerate the OS's normal scheduling activities. Hence, within even one single block when secrets are in plain text in the memory, the adversary could interrupt the enclave's execution and launch speculative execution attacks to leak all secrets in memory. In this case, any detection policy will fail if it accepts at least one interrupt.

To address this challenge, we suggest disallowing any interrupt during one invocation of an `ECall` function. This cannot be achieved by enclave developers alone. Instead, it requires support from the OS. So we have our first requirement for defense:

> **D1**: All interrupts should be eliminated during enclave's execution with the help from the OS. Basically, the OS has to provide extra supports, including but not limited to disabling timer interrupts and allocating enclave memory in advance to avoid potential page faults.

Consider that enclave programs are usually designed to execute minimal critical functions, disabling interrupts for a short period of time, *e.g.*, one second, should be acceptable for most platform for better security.

Note that the OS is untrusted even if she agrees to provide these supports. Hence, the enclave has to verify that these supports are in place, *i.e.*, by checking whether any interrupt has occurred during the execution. This can be done by setting the detection policy such that no AEX is allowed. Now we can see that an `ECall` function should be implemented as shown in Algorithm 1. Line 1 and line 4 are used together for verifying **D1**. Any interrupt occurred in between will be detected since the marker in the SSA will be overwritten during any AEX. Line 2 runs the co-location test introduced in HyperRace to ensure that the sibling hyper thread is also under control of the enclave. Line 3 executes operations specified by the original `ECall` function. A report of the outputs will be generated in line 5 if no speculative-execution attacks are detected during the execution. The report is used for attestation, which will be explained in further details in Sec. IV-B.

**Confidentiality between different invocations.** Now we will address **R3**. While secrets can be protected within one invocation, we still need to protect them from being attacked during the intervals between consecutive invocations if they are re-used. For example, after remote attestation, a shared ECDH key used to protect the communication between the remote party and the attested enclave will be generated and stored in the enclave memory. Before the invocation of the next `ECall` function, the shared ECDH key will be a natural target. The adversary could launch speculative-execution attacks to leak the ECDH key without being noticed. For example, Foreshadow could load the secret to L1 cache using `EWB` and `ELD` without call into any `ECall` function. Hence, we need a way to protect such secrets between different invocations.

The challenge here lies in that all content of enclave memory during the intervals can be learnt by the adversary. While encryption could help protect secrets, the encryption key itself is also a secret to be protect. To cover these intervals, we propose to leverage the sealing mechanism to protect secrets. The key observation is that the seal key is generated from within the CPU core and thus there is no need to store it in the memory. Hence, we could introduce the following mechanism:

*Sealing-protected intervals*: when an `ECall` function exits, all secrets are encrypted using the seal key and the seal key itself is cleared from the enclave memory.

Algorithm 2 demonstrates an implementation of sealing-protected intervals, which is supposed to replace line 3 in Algorithm 1, so that the sealing and unsealing processes can also be monitored and any speculative-execution attack during these processes can be detected. Here secrets can be unsealed from previously sealed copy (line 1 – line 3) or generated within the enclave (line 5). Line 6 executes operations specified by the original `ECall` function. Then the secrets need to be sealed again and the seal key needs to be cleared (line 7 – line 9).

Note that each time a new seal key (with a new 64-

**Algorithm 2:** Sealing-protected intervals

```
Input: Inputs (might include sealed secrets)
Output: Outputs
if sealed secrets is in Inputs then
    secrets ← Unseal(sealed secrets);
    Clear the seal key;
else
    secrets ← Gen_Secrets();
Outputs ← ECall_Func(Inputs, [secrets]);
Derive a seal key with a new random KeyID;
Seal secrets with the seal key;
Clear the seal key;
return Outputs
```

bit KeyID) should be used to prevent the adversary from decrypting the secrets using previously leaked seal key.

### B. Attestation Enhancement

Now that the enclave could ensure that it has processed the secrets during the execution without being attacked, it is still challenging to convince the remote party that the execution results are trustworthy if the local attestation and/or remote attestation mechanisms are compromised by the adversary. The adversary could learn the report keys (which are used to sign reports for the local attestation) and/or the EPID member private keys (which are used to sign the quotes for the remote attestation). Even if the enclaves could detect that they are attacked, the adversary could fake any enclave and thus any result using the leaked report keys and/or attestation keys.

The root cause is that these keys will be exposed to memory. The report keys will be derived and exposed to the target enclave's memory for verification. The remote attestation key will be exposed to the quoting enclave's memory to sign the attestation data. This provides the adversary a chance to obtain the keys.

To address this problem, we suggest the manufacturer to extend the functionality of SGX instructions to protect the attestation keys of both local attestation and remote attestation processes, and thus enable the remote party to detect attacks. The proposed extension of SGX for securing local attestation and remote attestation is described as follows:

**Securing local attestation.** We first deal with local attestation. When an attested enclave tries to attest itself to a target enclave, it calls EREPORT to sign its attestation data. EREPORT instruction derives the report key of the target enclave and signs the report data. Note that this process is done within the attested enclave using microcode, so that the target enclave's report key is not exposed to the attested enclave's memory. However, when the target enclave tries to verify the attestation data, it uses EGETKEY to derive its report key to verify the signature. In this procedure, the target enclave's report key is exported to the target enclave's memory, offering the adversary a window to steal the report key. Once the adversary obtains the report key of a target enclave, she could pretend to be any enclave to gain the trust of the target enclave. For example, with the leaked report key of Intel's quoting enclave, a malicious program could pretend to be any valid enclave to deceive the Intel's quoting enclave to pass the local attestation and then the remote attestation.

To address this problem, we propose the second requirement:

> **D2**: The report key should never be exposed in the enclave memory. Instead, to verify the report without exposing the report key, we propose a new SGX leaf instruction, named as EVERIFY. It takes as input a report and the corresponding report signature, derives the report key, verifies report signature, and outputs only the verification result (*e.g.*, a single bit indicating success or failure).

The extra overhead to implement EVERIFY using microcode should be low, because the main components, *i.e.*, deriving report key and generating the CMAC signature of the report, are already realized in EREPORT. With the introduction of EVERIFY, the report keys will never be exported to the enclave's memory and the local attestation can be secured.

**Securing remote attestation.** In the current SGX remote attestation design, the attestation key, *i.e.* EPID private member key, is exposed to Intel's quoting enclave and used to sign the attestation data, presenting the adversary a chance to extract the attestation key.

To secure the remote attestation procedure, we introduce the third requirement:

> **D3**: The remote attestation key should never be exposed in the enclave memory. We propose to extend the functionality of EREPORT to allow the derivation of a special report key using the root provisioning key only (without the root seal key), to sign the attestation data.

The overhead for extending EREPORT can be even smaller than implementing EVERIFY. Note that in this way the original EPID signature scheme is replaced by the message authentication scheme CMAC. Whether it is feasible to implement a signature scheme, or even an EPID-like group signature using microcode is beyond the scope of this paper.

To address the concerns that might rise from switching from EPID to CMAC. We would like to discuss the trade-offs:

- Privacy: EPID scheme leaks only group ID of the underlying SGX platform to both Intel and the remote client, while CMAC scheme leaks the underlying SGX platform's identity to Intel (as Intel has to know the exact identity to fetch the corresponding attestation key for verification) but hide the identity from the remote client completely (as the attestation signature can be encrypted using Intel's public key). In cases Intel is fully trusted, CMAC might provide better privacy protection against the remote client.
- Overhead: EPID scheme uses costly bilinear maps. Switching to CMAC might benefit applications that require frequent attestation. On the other side, it introduces extra workload for Intel's Attestation Service to maintain much

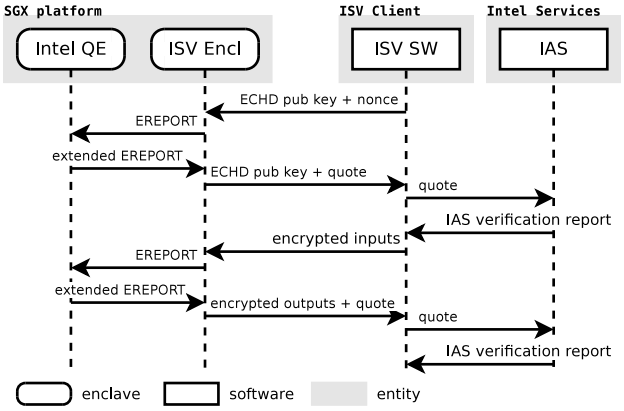


Fig. 1. Workflow of an example enclave application.

larger amount of report keys of all SGX platforms than EPID group public keys.

**Putting it together.** With the above extensions to secure both local and remote attestation, we can mitigate the attacks that can read the entire memory. Fig. 1 is the workflow of an example enclave application and the remote ISV client:

- The ISV client generates an ECDH key pair, sends the ECDH public key and a nonce to the SGX platform.
- The SGX platform launches the ISV enclave, the ISV enclave generates an ECDH key pair, derives the shared ECDH key, within one invocation. These secrets will then be sealed using a new seal key generated using a random `KeyID`. If no AEX is detected, the enclave produces a report with its ECDH public key and nonce (encrypted using the shared key) as report data using `EREPORT`. If any AEX is detected, the enclave terminates and no valid report will be generated.
- The report is passed to Intel's quoting enclave and verified by `EVERIFY`. The quoting enclave then signs the report via `EREPORT` using report key derived from the root provisioning key, resulting in a quote.
- The signed quote is then returned to the client, and forwarded to Intel for verification. A secure communication channel is created between the client and the enclave.
- If the verification passes, the client could then send the encrypted (using the shared key) secrets to the enclave for processing.
- The secrets are decrypted and processed, and the results are encrypted, within one invocation. Note that the shared ECDH key is unsealed within the invocation for decryption and encryption, and re-sealed after these operations. If no AEX is detected, the ISV enclave produces a report for the encrypted results. The report is signed by Intel's quoting enclave and returned to the client along with the encrypted results. If any AEX is detected, the ISV enclave terminates and no valid report can be generated. Note that the sealed ECDH key is within the enclave memory and will be gone when the enclave terminates. Hence, re-launching the enclave after attack will not resume the execution.
- After verification by Intel, the client could assure that the results are reliable and the secrets are not leaked.

Note that the sealing process mentioned here is used for protecting secrets between different invocations of the same enclave instance, rather than persisting secrets from one enclave instance to anther. Because the adversary could attack one enclave instance to obtain the seal key and then forge arbitrary sealed data to fool the other enclave instances. To enable the original functionality of sealing, we need to address such replay-assisted speculative-execution attacks.

### C. Replay Prevention

Intel SGX has already provided trusted monotonic counters through trusted platform service to address replay attacks. However, in its current design, the trusted platform service itself is vulnerable if memory confidentiality is breached:

- Counter values in sealed data can be manipulated: seal keys of the ISV enclave might be obtained by the adversary to forge valid sealed data with up-to-date counter value to bypass the monotonic counter check.
- Counter values in the CSME can be manipulated: The adversary might attack the platform service enclave (PSE) that provides the trusted monotonic counter service, to obtain the credentials for communication with the Intel Converged Security and Management Engine (CSME), which manages the replay-protected storage. In this way, the adversary could pretend to be the PSE to communicate with the CSME to manipulate the monotonic counter values to launch replay attacks.

To address the first issue, we leverage local attestation to ensure the integrity of counter values. Specifically, the enclave could generate a local attestation report using `EREPORT` with itself as the target enclave, and then verify it using `EVERIFY`. Since the report key will not be revealed in both processes, the adversary could not fake a valid local attestation report, so that the integrity of the counter values can be guaranteed.

The second issue can be addressed similarly as how we enhance remote attestation. Here, the CSME can be considered as the remote client the PSE would like to attest itself to. The PSE uses the extended `EREPORT` to generate a report with a special report key (for replay prevention purpose) derived using the root provisioning key only. The same report key can be derived by Intel also. The CSME needs to communicate with Intel's Provision Service to obtain this report key to verify the generated reports.

**Secure sealing processes.** With the above replay prevention, we now describe how to secure the sealing process:

- Within one invocation that performs initial sealing, the enclave increases the monotonic counter value first. Then, a new seal key with a randomly generated `KeyID` is derived to seal secrets. After sealing, the seal key is cleared. If no attack is detected then, generate a report of the sealed data to be attached to the sealed data.
- Within one invocation that perform unsealing and re-sealing, the enclave also increases the monotonic counter

value first, and then verifies the attached report and check whether the sealed counter value is less than the current monotonic counter value by 1 (due to the increment at the beginning). If all checks pass, it unseals the secrets, processes the secrets and re-seals the secrets with a new seal key. If no attack is detected in the end, it generates a report and attaches it to the sealed data.

## V. Security Analysis

In this section, we analyze the security of our proposed scheme. We argue the following secrets can be protected:

- *Intel's secrets*: Intel secrets include remote attestation keys and report keys of Intel issued enclaves, which are used to secure the attestation loop between the ISV enclave and the remote client. When the adversary could obtain such secrets, she could disguise as any valid enclave.
- *ISV enclave secrets*: ISV enclave secrets include random generated secret keys, seal keys, report keys and remote user's secrets to be processed by ISV enclaves.

For Intel secrets, since they are so critical that any leakage of them could break SGX's security guarantees completely. We proposed **D2** and **D3** to prevent them from being exposed to the enclave memory, thus eliminating the chances of speculative-execution attacks on them completely. In this way, the adversary could never forge a valid quote. When the remote client received a quote that is verified to be valid by Intel Attestation Service, she could confirm that the quote data is generated by the specific ISV enclave.

For ISV enclave secrets, when the remote client receives a result from the ISV enclave with a valid quote, all ISV enclave secrets that ever appear in the enclave memory cannot be leaked via speculative-execution attacks. Since the adversary might launch speculative-execution attacks at any time, each result received from the SGX platform should be verified via remote attestation. With **D1** and HyperRace in place, speculative-execution attacks during an `ECall` invocation will be detected and no valid quote will be generated. To avoid being detected, the adversary could only attempt to launch attacks between invocations. Since secrets are protected by sealing between invocations, the adversary has to obtain the seal key that is exposed to enclave memory. In our scheme, a random new seal key is generated each time, which is protected by the following means:

- Seal keys used for sealing-protected intervals: each seal key is exposed to the enclave memory twice, once for sealing and the other for unsealing. Since the seal key is cleared before finishing an `ECall` function invocation, it can only be attacked during one invocation, which will be detected by the end of the corresponding `ECall` function invocation and thus no valid quote will be generated.
- Seal keys used for persistent storage in general: The detection of speculative-execution attacks after the sealing operations guarantees that the seal key is secure after it is generated and used for sealing. As our scheme associates a monotonic counter with each sealed data to make sure the seal key is only used once for sealing and once for unsealing, its security can be guaranteed. The unsealed data should only be trusted if a valid report is attached.

With all the analysis above, we can conclude that no adversary could learn enclave secrets without being detected.

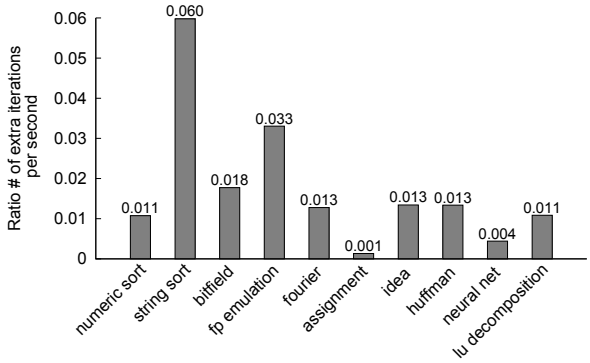


Fig. 2. Performance gain.

## VI. Implementation and Evaluation

We implemented a prototype system on Linux kernel 4.4.40 and Intel SGX SDK (version 2.2.100.45311). Our evaluation focuses on **D1**. Since **D2** and **D3** rely on the manufacturer, we leave their evaluation to future work.

### A. Implementation

To achieve **D1** which requires the OS to eliminate interrupts for logical cores running in the enclave mode, we made two modifications to the kernel: firstly, we set the scheduler option `isolcpus` in the Linux kernel configuration to isolate logical cores the might run enclave code from the kernel scheduler so that the OS will not schedule other tasks on the logical core that runs the enclave code to trigger interrupts. Second, to eliminate local timer interrupts, we implemented a kernel module to provide APIs for configuring the local Advanced Programmable Interrupt Controller (APIC). Specifically, we disabled and enabled the APIC using the APIC software enable/disable flag in the spurious-interrupt vector register, according to Intel manual [24]. Hence, before entering the enclave's critical sections, the local APIC will be disabled via the kernel module. After leaving the enclave's critial sections, the local APIC will be enabled again.

We implemented the co-location test and AEX detection mechanism introduced in HyperRace [20] as a shared library to provide APIs to enclave developers.

### B. Evaluation

We evaluated the performance on a Dell Latitude 5480 laptop with an Intel Core i7-7820HQ processor and 8GB memory. We ported *nbench*, a set of lightweight CPU and memory performance benchmarks, to run inside SGX enclaves. *nbench* includes 10 benchmark applications. It measures the number of iterations to perform each benchmark per seconds.

Compared to the performance of *nbench* without our protection, our evaluation shows that the performance of the

protected application is actually improved. Fig. 2 shows the performance gain, reflecting the extra percentages of iterations that could be performed per second with **D1** and our protection. The performance gains for the 10 benchmark applications range from 0.1% to 6%, with a geometric mean of 1.8%. The improvement results from two aspects: (1) since AEX is eliminated, the cost of context switches is saved, (2) there is no need to periodically detect AEXs during one invocation (periodical AEX detection contributes most of the overhead in the original HYPERRACE design); only one co-location test is needed at the beginning of an invocation.

## VII. CONCLUSION

In this paper, we propose to extend HYPERRACE to detect speculative-execution attacks. The proposed mitigation scheme requires supports from untrusted OS but is able to verify OS's behavior. The security of our proposed solution is analyzed. Our implementation of an SGX shared library and OS modifications for attack detection is evaluated.

## ACKNOWLEDGMENTS

The work was supported in part by the NSF grants 1718084 and 1750809.

## REFERENCES

[1] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud Using SGX," in *IEEE Symposium on Security and Privacy*, 2015.

[2] F. Zhang, E. Cecchetti, K. Croman, A. Juels, , and E. Shi, "Town Crier: An Authenticated Data Feed for Smart Contracts," in *23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[3] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *25th USENIX Security Symposium*, 2016.

[4] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *14th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2017.

[5] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014.

[6] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016.

[7] R. Strackx and F. Piessens, "Ariadne: A Minimal Approach to State Continuity," in *USENIX Security Symposium*. USENIX, 2016.

[8] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," in *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016.

[9] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *2017 USENIX Annual Technical Conference*, Santa Clara, CA, 2017.

[10] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux Applications With SGX Enclaves," in *Network and Distributed System Security Symposium*, 2017.

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

[12] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Stealing intel secrets from sgx enclaves via speculative execution," in *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, June 2019.

[13] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium*, 2018.

[14] S. van Schaik, A. Milburn, S. sterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *Security and Privacy (SP), 2019 IEEE Symposium on*, May 2019.

[15] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. V. Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, "Fallout: Reading kernel writes from user space," *arXiv preprint arXiv:1905.12701*, 2019.

[16] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *ACM Conference on Computer and Communications Security*, 2019.

[17] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *International Symposium on Microarchitecture*, 2018.

[18] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *International Symposium on Microarchitecture*, 2018.

[19] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019.

[20] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018.

[21] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium*, 2017.

[22] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Report 2016/086, 2016, https://eprint.iacr.org/2016/086.

[23] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel Software Guard Extensions: EPID Provisioning and Attestation Services," Intel, Tech. Rep, Tech. Rep., 2016.

[24] "Intel 64 and IA-32 architectures software developer's manual, combined volumes:1,2A,2B,2C,3A,3B,3C and 3D," https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2017, order Number: 325462-063US.

[25] C. Percival, "Cache missing for fun and profit," in *2005 BSDCan*, 2005.

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

[27] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Aug. 2019.

[28] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium*, 2018.

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

[30] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy*, 2019.

[31] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *Computer Security – ESORICS 2019*. Cham: Springer International Publishing, 2019.

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

[33] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies*, 2018.

[34] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[35] B. Gülmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar, "Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning," *arXiv preprint arXiv:1907.03651*, 2019.