IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 14, 2021, accepted May 28, 2021, date of publication June 2, 2021, date of current version June 9, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3085395

Multibyte Microarchitectural Data Sampling and Its Application to Session Key Extraction Attacks

YOUNGJOO SHIN

School of Cybersecurity, Korea University, Seoul 02841, South Korea

e-mail: syoungjoo@korea.ac.kr

This research was supported by a National Research Foundation of Korea (NRF) grant, funded by the Korean government (MSIT) (No. 2020R1F1A1065539). This work was supported by an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2019-0-00533, Research on CPU vulnerability detection and validation). We received support in the form of a Korea University Grant in 2020.

ABSTRACT Microarchitectural data sampling (MDS) attacks leak secret data from the internal buffers of a processor to the attacker during transient execution. Because of the narrow window of transient execution, previous MDS attacks relied on repetitive sampling to obtain arbitrarily sized data from the buffer. However, as an MDS attacker cannot control the address for data leakage, such an approach significantly degrades the signal-to-noise ratio in the sampled data. In this paper, we propose a novel multibyte microarchitectural data sampling technique for performing MDS attacks. The proposed technique allows several continuous bytes to be captured in one execution without repetition of sampling. The implementation of the technique is quite challenging, because a transient execution window is not sufficiently large to allow multibyte sampling to be completed. We address this problem by leveraging a return stack buffer-based speculation technique, which originally was used for variants of Spectre-type attacks. We repurpose it to enlarge the transient execution window in our attack. Our implementations can capture data of up to 16 bytes in length in one execution from a line-fill buffer. To validate the effectiveness of the multibyte sampling technique, we demonstrate session key extraction attacks against secure network protocols. In particular, our objective is to extract AES-128 and AES-256 keys from TLS and SSH applications. To recover session keys in a postprocessing phase efficiently, we also propose a novel clustering-based search method that assembles the bytes of interest from the noisy sampled data. The experimental results show that our technique can successfully extract AES-128/256 session keys from victim applications with a probability of at least 98% and a reasonable search complexity.

INDEX TERMS Microarchitectural data sampling, transient execution attack, session key extraction attack.

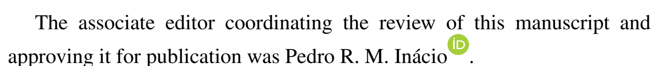
I. INTRODUCTION

Modern processors aggressively utilize out-of-order and speculative execution techniques in their microarchitecture design to maximize instruction-level parallelism (ILP). However, pursuing performance without considering security ultimately leads to the occurrence of the well known transient execution attacks, such as Meltdown [1] and Spectre [2].

Transient execution attacks exploit the vulnerability of ILP optimization techniques to expose secret data that would otherwise be protected from unauthorized access. Specifically, an attacker initiates transient executions, where the protected data are illegally loaded from memory and encoded to microarchitectural states in a cache. The data are then delivered to the attacker through a cache covert channel.

As the execution of instructions is canceled quickly by the processor, the time window of transient execution for successful data leakage is extremely short. Hence, previous implementations of Meltdown- and Spectre-type attacks attempted to leak only one byte per execution rather than capturing a full byte of the data at time. On the basis of such byte-by-byte leakages at continuous memory locations, the attacker can obtain any arbitrarily sized data.

Recently, new vulnerabilities have been found on microarchitectural buffers inside some Intel processors [3], [4]. The internal buffers are designed to keep data corresponding to in-flight load and store instructions temporally. These vulnerabilities allow an attacker to read stale data remaining on

The associate editor coordinating the review of this manuscript and approving it for publication was Pedro R. M. Inácio.


This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

the internal buffer. This opens a brand new type of attacks [5]–[9], which are referred to as microarchitectural data sampling (MDS) attack.

An MDS attack is initiated with a load instruction that causes certain faults or microcode assists. The vulnerable processor may transiently execute subsequent instructions with stale data which is forwarded due to lazy exception handling. The leaked data are then recovered later through the cache covert channel.

Although the MDS attack shares the basis of the transient execution technique with the previous Meltdown-type attacks, the essential difference is that all the data leaked by the MDS attack are agnostic to the memory address [5]. That is, an MDS attacker has no control over the full memory address at which he/she wants to leak. In-flight data sampling without knowledge of the memory address results in data leakage irrespective of the secret data of interest.

For this reason, the sampling of one byte of data per execution as in the byte-by-byte approach of Meltdown-type attacks is not effective for MDS attacks in terms of feasibility and performance. For instance, the leakage of a secret value with 16 bytes in length (*e.g.,* an AES-128 key) requires at least 16 consecutive runs of the one-byte sampling technique. In fact, orders of magnitude more runs are necessary in practice because of the address-agnostic property of the MDS attack itself, as well as various types of system noise, the source of which is the non-deterministic behavior of operating systems. When the raw data has been collected from sampling, each byte of the secret must be distinguished from irrelevant data. Recovering the secret by filtering out unrelated bytes is not trivial, and may degrade the performance and feasibility of MDS attacks, as we discuss in Section IV. Previous work attempt to improve the one-byte sampling by proposing a technique that samples 3-byte data at a time [5], [8]. However, the sampling length is still not long enough to overcome this problem.

In this paper, we address these challenging problems and introduce more effective and practical MDS attacks. In particular, we propose a multibyte data sampling technique that captures multiple bytes of in-flight data at each execution. The key idea on which our approach is based is that the sampling of continuous bytes from internal buffers in one execution can exclude any irrelevant bytes and preserve the order of bytes at least among the leaked data. The properties of noise exclusion and order-preserving sampling result in a high signal-to-noise ratio in the data. Therefore, multibyte data sampling improves effectiveness and practicality of MDS attacks.

The challenging issue in the implementation of multibyte sampling is the completion of multiple-byte encoding, which clearly takes longer than one-byte encoding, within the narrow transient execution window. For this, we enlarge the window size by leveraging a return stack buffer (RSB) based speculation technique. This technique was originally used for some variants of Spectre-type attacks that leverage the RSB [10], [11]. We repurpose it to create a long delay in the resolution of the return address. Thus, we can obtain a sufficiently large transient execution window for the multibyte encoding.

Assisted by the window enlarging technique, we implemented our multibyte data sampling attack. More specifically, our implementations include MDS-64, MDS-96 and MDS-128, which are able to leak 8, 12 and 16 bytes at one execution, respectively. The implementations exploit the vulnerability of the line-fill buffer (CVE-2018-12130) [3], although our technique is not confined to this vulnerability. We conducted experiments to evaluate the performance of our implementations in terms of the throughput and success rate. As aforementioned, previous work already showed that it is possible to achieve leakage of a few bytes of data at a time in MDS attacks. The proposed multibyte technique significantly increases the sampling length by enlarging the transient execution window.

To validate the effectiveness of our multibyte sampling technique in practical applications, we demonstrate session key extraction attacks against the Transport Layer Security (TLS) and Secure Shell (SSH) protocols. In particular, our objective is to extract AES-128 and AES-256 session keys by using the multibyte sampling technique against victim applications running these protocols. The construction of TLS and SSH applications is based on the recent versions of the OpenSSL and wolfSSH libraries, respectively, both of which support AES hardware acceleration (*e.g.,* AES-NI). We present an MDS attack that successfully leaks AES-128 and AES-256 keys from those applications.

We also present our novel clustering-based search method for recovering AES-128/256 session keys from raw data. The proposed method allows the secret bytes to be assembled successfully from noisy values. The use of this method in combination with an exhaustive search significantly improves the success probability of key recovery during postprocessing. The experimental results show that with a probability of at least 98% and reasonable search complexity, we can extract AES-128/256 session keys from the raw data sampled for 5 min in an attack against TLS and SSH applications.

To mitigate MDS attacks, including that proposed in this paper, we present several countermeasures. Specifically, we discuss some solutions to protect MDS attacks including hardware-based mitigation, isolation of the leakage source, encryption of the protected data, and attack detection in this paper.

The remainder of this paper is organized as follows. In Sections II and III, we provide related work and background knowledge on the proposed attacks, respectively. In Section IV, we describe in detail the proposed technique for multibyte MDS attacks, including motivation, implementation and evaluation. In Section V, we present session key extraction attacks against the TLS and SSH protocols by using the multibyte data sampling attack. In Section VI, we present several countermeasures to the proposed attack. Finally, we conclude this paper by summarizing our work in Section VII.

II. RELATED WORK

A. MICROARCHITECTURAL DATA SAMPLING ATTACKS

The security vulnerabilities in the memory subsystem of modern processors have been exploited to implement various MDS attacks.

RIDL [6] and ZombieLoad [5] are the first attacks that leverage a line-fill buffer (LFB) and load ports as leakage sources on Intel processors. They demonstrate several attacks that leak secret data such as AES-128 keys and passwords from victim applications.

Fallout [7] exploits a store buffer as a leakage source to leak the data corresponding to recent writes. It demonstrates attacks for recovering secret data such as AES-128 keys written by kernels as well as for breaking kernel address space layout randomization (KASLR) used in the OS.

Moghimi *et al.* [8] performed an in-depth analysis on Meltdown-type attacks via a fuzzing-based approach, and consequently discovered new variants of MDS attacks named Medusa. Medusa allows an attacker to leak secret data from specific implicit write-combining memory operations such as fast string copies. As a case study, an RSA signing key recovery is demonstrated by using the attack.

Ragab *et al.* [9] implemented an x86 instruction profiling tool named CrossTalk to investigate a variety of complex microcoded operations. As a result of the investigation, they discovered a cross-core MDS attack that leaks secret data on a staging buffer which is shared between all cores.

All the above attacks share the structure of one-byte data sampling. There are already some improvements that sample up to 3 bytes of data at a time without increasing the transient window [5], [8]. However, they are not enough for achieving more efficient and practical MDS attacks. Our study is aimed to address the limitations of the previous techniques due to the narrow window. We basically extend the sampling capacity with the help of the multibyte data sampling technique. We validate its feasibility by demonstrating an attack that extracts AES-128/256 keys from practical applications.

B. OTHER MICROARCHITECTURAL ATTACKS

We also present additional microarchitectural attacks that are related to our work.

1) MELTDOWN- AND SPECTRE-TYPE ATTACKS

According to the Canella *et al.*'s taxonomy [12], transient execution attacks are classified into Meltdown- and Spectre-type attacks. Meltdown-type attacks exploit the out-of-order execution of ILP optimization techniques in modern processors. Transient executions of Meltdown are initiated by raising certain exceptions. In most studies, a page fault exception is used for delivering the attack; the fault is raised by privilege violation [1], R/W access violations [13] and the absence of the present bit in a page table entry [14], [15]. Some Meltdown-type attacks utilize other exceptions such as a device-not-available exception [16] and a general protection fault [17].

Spectre-type attacks, however, exploit a speculative execution technique. Specifically, a malicious transient execution is initiated by inducing misspeculations on branch instructions. Various types of prediction units are used for this attack including the branch target buffer [2], memory order buffer [18] and pattern history buffer [2], [13], [19]. Other variants of Spectre-type attacks make use of an RSB to create misspeculation-based transient executions [10], [11].

2) CACHE TIMING ATTACKS

Cache contention allows cache timing channels through which an attacker can learn secret information from victim applications [20]. Cache timing attacks are demonstrated on a wide variety of target applications. Most work show their attacks against a number of cryptographic implementations; target applications include public key-based algorithms such as RSA [21]–[24], ECDSA [25], ECDH [26], [27] and ElGamal [28] as well as symmetric key-based algorithms such as AES [29]–[32].

The security implication of cache timing attacks is not confined to cryptographic implementations. Several work present their attacks against various (non-cryptographic) practical applications such as leaking sensitive data from input devices [33], [34] and inferring information of a browsing history [35] and firewall policies [36], [37].

Cache is not the only resource that has a timing channel between an attacker and victim. In simultaneous multi-threaded (SMT) processors, various hardware resources are shared between two logical threads, thus other timing channels may be potentially created. Actually, recent work presented microarchitectural attacks exploiting timing channels existing on an execution port [38], translation look-aside buffer [39], [40], memory order buffer [41], [42] and way predictors [43].

III. BACKGROUND

In this section, we provide background knowledge of MDS attacks.

A. OUT-OF-ORDER AND SPECULATIVE EXECUTION

Processors maximize instruction-level parallelism by utilizing various optimization techniques. Two of the techniques are *out-of-order* and *speculative* executions. The out-of-order execution technique allows a micro-operation (μ-op) to be executed as soon as its resources (*e.g.,* source operands or execution units) become available, regardless of the program order. However, the retirement of μ-ops, after which the execution result becomes visible in the architectural state (*e.g.,* registers or memory), is processed in the program order. The reorder buffer (ROB) tracks in-flight μ-ops and ensures that they are retired in order.

Speculative execution is a second optimization technique for managing branch instructions. It enables the processor to jump to a predicted location speculatively before the actual destination of the branch instruction has been determined. Successful speculation would achieve a substantial

performance gain. If the speculation fails, however, the processor has to squash all μ-ops in the pipeline and then restart its execution at the correct destination. The branch prediction is conducted with the help of dedicated hardware, such as a branch prediction unit (BPU) and branch target buffer (BTB). The RSB is also utilized for branch instructions such as `call` and `ret` designated for the procedure call mechanism.

B. CACHE COVERT CHANNEL

A cache is an integrated hardware component in a processor for providing low-latency memory access. Cache contention between processes or threads incurs a cache covert channel, by which information can be illegally transmitted from one side to another.

The Flush+Reload technique [21], [34] is typically used for the cache covert channel. In this technique, a sender and receiver use a shared array as a transmission medium. We refer to this array as an F+R buffer in the rest of this paper. The F+R buffer consists of 256 elements, each of which is at least more than 64 bytes in size. These elements are initially evicted from the cache: the `clflush` instruction in a x86 architecture can be utilized to evict specific cache lines.

A sender who wishes to transmit a byte x ($0 \leq x \leq$ 0xff) to a receiver encodes x into the F+R buffer. That is, the sender performs a memory access to the x-th element in the buffer to bring it into the cache. Then, the receiver determines which element has been accessed by the sender through timing analysis. A longer access time indicates that the element comes from the memory, whereas a shorter access time indicates that it comes from the cache. Thus, the receiver can decode byte x from the F+R buffer.

C. TRANSIENT EXECUTION ATTACKS

A *Transient instruction* refers to an in-flight instruction that is executed but then disappears from the ROB without being retired. The execution of the transient instruction is never in fact committed. However, it may affect the microarchitectural state of the processor, especially the cache, which remains visible even after the transient instruction has vanished.

Transient execution attacks exploit the out-of-order and speculative execution techniques to realize a malicious execution of transient instructions. These instructions cause an unauthorized memory access to protected data, leaving an elaborate footprint on the cache according to the value of the data. The protected data, which have been encoded in the cache, can be delivered to an attacker through a cache covert channel.

Based on Canella *et al.*'s taxonomy [12], transient execution attacks are classified into Spectre and Meltdown types according to the type of ILP optimization technique (*e.g.,* out-of-order or speculative execution) that is exploited. Spectre-type attacks exploit transient execution by creating a misspeculation on branch predictors, whereas Meltdown-type attacks exploit the fact that, when a preceding instruction encounters an exception, subsequent instructions may be transiently executed out of order. In Meltdown-type attacks, the exception is typically caused by load instructions that would result in a page fault or microcode assist. Unlike in Spectre-type attacks, an exception should be addressed appropriately by handling or suppressing it during the execution of Meltdown-type attacks.

D. MICROARCHITECTURAL DATA SAMPLING VULNERABILITIES

Modern processors are equipped with several microarchitectural buffers inside their cores to serve load and store instructions. These buffers are typically shared by logical threads from the same or even different security domains. Some Intel processors have security flaws in the memory subsystem including microarchitectural buffers [3], which lead to MDS attacks as follows.

- Microarchitectural Fill Buffer Data Sampling (MFBDS, CVE-2018-12130): An LFB is a temporal buffer between an L1D cache and lower memories. The LFB handles a demand load request that missed the L1D cache and temporally holds the requested line before serving it to the cache. MFBDS vulnerability allows an attacker who mounts transient execution attacks to leak stale data on the LFB from other threads that would otherwise be protected.
- Microarchitectural Load Port Data Sampling (MLPDS, CVE-2018-12127): A load port is a temporal buffer that is used when loading data into registers. This vulnerability allows an attacker to leak stale data on the load port by mounting transient execution attacks.
- Microarchitectural Store Buffer Data Sampling (MSBDS, CVE-2018-12126): A store buffer temporally holds the data of store operations and may serve subsequent loads if their addresses match. This vulnerability allows an attacker to leak stale data belonging to preceding store instructions.

Similarly to Meltdown-type attacks, MDS attacks exploit load instructions causing a page fault or microcode assist. The load keeps its execution with the stale data which is forwarded from one of the microarchitectural buffers mentioned above. Subsequent transient instructions created by the faulting load (or microcode assist) deliver the data via the cache covert channel. Compared to the previous Meltdown-type attacks, however, the control of the memory address for leakage is more restrictive in MDS attacks. Therefore, MDS attacks are more likely to suffer from noise due to the leakage of unrelated data.

IV. MULTIBYTE MICROARCHITECTURAL DATA SAMPLING

A. MOTIVATION AND GOALS

Prior to describing details on the proposed method, we present motivation and our goals in this section.

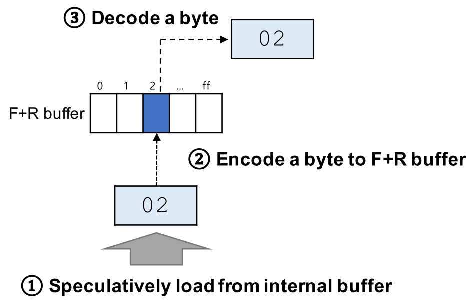


FIGURE 1. Leaking one byte of secret data by microarchitectural data sampling.

1) MOTIVATION

In general, MDS attacks are conducted in three steps (Fig.1). In the first step, a transient load of in-flight data from a microarchitectural buffer is executed by inducing a fault or microcode assist. In the second step, the leaked value is encoded into an F+R buffer; the byte is translated to the offset of the F+R buffer and the corresponding cache line is loaded into a cache. Finally, the value is decoded by using the Flush+Reload technique in the third step.

Note that the first and second steps in MDS attacks are performed in the transient execution domain. As their execution is quickly canceled by the processor due to the narrow transient window, the execution path should be minimized as much as possible. This significantly limits the data leakage channel's capacity.

In fact, previous Meltdown-type attacks (*e.g.,* Meltdown [1] and Foreshadow [14]) also suffered from the limited capacity of the leakage channel. However, they could fully or partially control the memory address of the data of interest. Therefore, it was possible to reliably obtain all the bytes of a secret through byte-by-byte leakage by means of iteration at continuous target addresses.

Compared to the previous Meltdown-type attacks, however, MDS attacks have less control over the address of the secret. This limitation leads to leakage of a significant amount of dummy data irrespective of the secret. Hence, it is challenging to recover the whole secret from sporadic in-flight loads by iterations of sampling within the narrow transient execution window. To achieve reliable leakage of a secret in MDS attacks, several techniques were applied in previous studies, as follows.

- *Synchronization with victim* [5]: In this technique, the spy (*i.e.,* the MDS attacker) invokes data sampling only when the victim executes a specific function that loads the data of interest (*e.g.,* AES encryption). This way, most unrelated dummy data can be filtered out. Although this technique is practical in some threat models such as SGX [44], it is restrictive as tight synchronization of the victim with the spy is necessary.
- *Utilizing prior knowledge of the secret* [6]: In this technique, the spy filters out noise by leveraging prior knowledge about the secret. Specifically, the spy computes XOR-masking of the leaked data with previously obtained bytes of the secret in the transient execution. Masking with any irrelevant bytes results in misalignment to an F+R buffer, and thus causes no microarchitectural changes. The prior information given to the spy includes several fixed strings in the victim's executable or partial bits of the secret previously obtained by other means.
- *Leaking overlapping data* [5]: The spy additionally captures the existing data called a *domino* byte. The domino byte contains partial bits of adjacent secret bytes with which leaked data can be distinguished from noise. In this technique, at least one domino byte is needed to leak two bytes of a secret.

Although the above techniques may strengthen the sampling capacity to some extent, their effectiveness is confined to limited attack scenarios. Specifically, synchronization with the victim is effective only under strict conditions that require sharing of the victim's physical memory and a deep knowledge of the victim's executable binary. For instance, in the cross-VM attack model, the memory sharing needs deduplication support from underlying hypervisors, which is usually disabled or no longer available in most commercial products for security reasons. An additional technique that leverages prior information of a secret may not be applicable to certain attack scenario such as a session key extraction attack described in the following section. For the domino technique, the reading of an additional domino byte to leak a two-byte secret even consumes the narrow bandwidth of the leakage channel, significantly degrading the attack performance.

2) OUR GOALS

Motivated by the limitations and problems of the existing techniques, we attempt to achieve more effective and practical microarchitectural data sampling attacks. In particular, our goals in implementing our data sampling attack are threefold, as follows.

- *No synchronization with victim*: Our attack should efficiently leak secret data from a victim without the need for synchronization.
- *No prior knowledge of the secret*: Our attack should efficiently leak secret data from a victim even if no prior information of the secret is known.
- *Improvement in the channel capacity*: Our attack should efficiently leak secret data with a higher throughput by means of multibyte data sampling.

B. MULTIBYTE DATA SAMPLING

1) BASIC APPROACH

The key idea of achieving the above goals is to attempt to capture all the bytes of the secret data at one run. To this end, we devise a novel multibyte data sampling technique.

Fig.2 illustrates the manner in which our technique can obtain multiple bytes of a secret at each execution. Multibyte sampling is conducted in three steps in the same manner as the original data sampling. In the first step, N-byte secret data

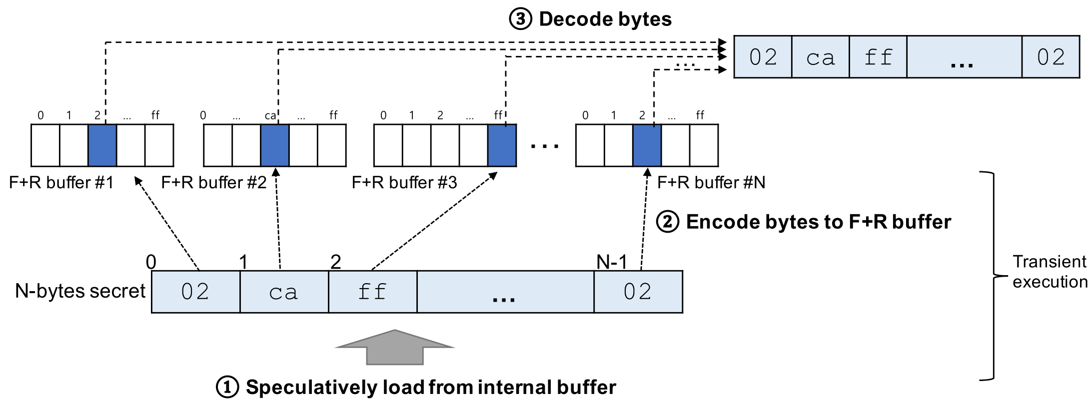


FIGURE 2. Multibyte microarchitectural data sampling.

are transiently loaded to a register from the internal buffer. In x86-64 architecture, the operation can load up to a 64-bit word (*i.e.,* N=8) to a general purpose register. In the second step, the loaded data are encoded by using multiple F+R buffers. Specifically, each byte of the data is encoded into the corresponding buffer. The secret is finally recovered in the third step by sequentially probing the F+R buffers.

2) CHALLENGING PROBLEM AND SOLUTION

The multibyte data sampling basically extends the existing sampling technique by increasing its encoding length to multiple bytes. However, it is not trivial to increase the length of data for encoding. As described in the previous section, the encoding step is executed in the transient domain, which has an extremely small time window. Because it takes longer to encode multiple bytes of data than a single byte, the completion of this step within the narrow execution window presents a challenge.

We solve this problem by enlarging the window size sufficiently to achieve reliable multibyte encoding. In particular, we utilize an RSB-based speculation technique [16], [45]–[47] for our multibyte data sampling attack. This technique was originally used for some variants of Spectre-type attacks that leverage the RSB. We repurpose it to enlarge the transient execution window. RSB is a branch prediction unit designated for a procedure mechanism in an x86 architecture. It caches a return address for a `call` instruction and then serves fast lookup to the subsequent `ret` instruction for the return address. The RSB-based speculation technique creates transient execution by modifying the return address in the stack to another destination so that misspeculation occurs. Most importantly, this technique can intentionally cause a delay in branching to the correct destination after the misspeculation. It allows a sufficiently large transient execution window for multibyte encoding to be built.

The code layout of our multibyte data sampling attack is illustrated in Fig.3. The first step in the execution of the code is a `call` instruction. It branches to the location of a function `func` while simultaneously saving the return address `retaddr` to the stack and to the RSB (Step (1) in Fig.3(a)). In the location next to the `call` instruction, which is located at `retaddr`, is the code that performs transient loading and encoding of the secret data. The function `func` manipulates the return address on the stack to be `new_retaddr`, which points to the location of the execution code that decodes the secret. This stack manipulation causes a discrepancy between the content of the stack and that of the RSB. The `ret` instruction speculatively branches to `retaddr` by consulting the RSB, but eventually branches again to `new_retaddr` (Steps (2) and (3) in Fig.3(a), respectively).

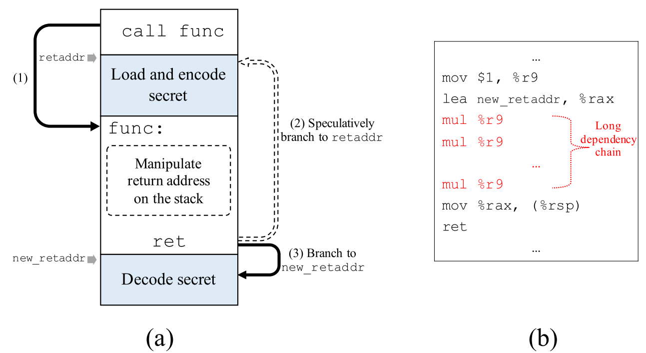


FIGURE 3. Code layout of the proposed multibyte microarchitectural data sampling attack.

As the transient execution of secret loading and encoding remains alive until `ret` terminates the misspeculation, we can increase its lifetime by delaying `ret` to determine the correct destination. As shown in Fig.3(b), this delay can be achieved by inserting a number of `mul` instructions to calculate the return address. The series of `mul`s sequentially multiply data on `%r9` with a multiplicand `%rax`, finally yielding the return address to `%rax` as output. The data dependency on the operands forces sequential execution of these instructions and allows a sufficient delay in determining the return address to be achieved.

3) IMPLEMENTATION

We now present a detailed description of our implementation for the multibyte data sampling attack. As an

```
1    clflush (%rbx)
2    call func
3    movq (%rcx), %rax
4    movzx %al, %r8
5    shl $12, %r8
6    movq (%rdi,%r8,1), %r9
7  .rept 7
8    add $0x100000, %rdi
9    shr $8, %rax
10   movzx %al, %r8
11   shl $12, %r8
12   movq (%rdi,%r8,1), %r9
13 .endr
14 func :
15   mov $1, %r9
16   lea FR_decode,%rax
17 .rept MULS
18   imul %r9
19 .endr
20   mov %rax, (%rsp)
21   ret
```

FIGURE 4. Implementation of a multibyte data sampling attack (N=8).

instance of the attack, the implementation attempts to leak eight bytes of a secret at a time (*i.e.,* N=8) by exploiting the LFB microarchitectural data sampling vulnerability (CVE-2018-12130) [5], [6].

Fig.4 shows the assembly code of the implementation. Prior to launching the attack, the code performs several initialization steps. First, a memory page is allocated and mapped to two virtual addresses v and k, which are for user space and kernel space, respectively. The code also prepares a number of F+R buffers (*i.e.,* eight buffers in this implementation), which are contiguously allocated in the address space. Each F+R buffer consists of 256 entries, each of which belongs to a different page, taking up a memory space of 1,024 KB in total. The execution of the assembly code begins with the initialization of several registers to specific addresses. The registers `%rbx` and `%rcx` have addresses v and k, respectively, and `%rdi` has an address of the first F+R buffer.

Each line of the assembly code is described as follows.

- **Line 1:** This instruction evicts an address v referred by `%rbx` from the cache so that the subsequent faulting load (line 3) can leak stale data from the LFB.
- **Line 2:** This instruction initiates RSB-based speculation by branching to a function `func` located in lines 14 ˜ 21.
- **Lines 3 ˜ 13:** These instructions belong to the transient execution domain, which is created by speculation from the RSB. The `%rax` register contains eight bytes of stale data leaked from the LFB. The least significant byte of `%rax` is encoded to the first F+R buffer, where its base address is referred by `%rdi` (lines 3 ˜ 6). The remaining bytes are then sequentially encoded to F+R buffers in the same manner as the least significant byte (lines 7 ˜ 13). The F+R buffer for a subsequent byte is located adjacent to the previous buffer. Hence, the base register `%rdi` can be adjusted by adding `0×100000` to the base address of the previous buffer (line 8).
- **Lines 14 ˜ 21:** These instructions constitute the body of the function `func`, where a return address on the stack is replaced with the address of the `FR_decode` function, which performs secret decoding (line 20). The return address is calculated by repetitive multiplications (*i.e.,* `mul` instructions) using two operands `%r9` and `%rax`. The number of `muls` (*i.e.,* `MULS` in line 17) is determined according to the length (N) of the sampling data and microarchitectural configurations of a target processor, such as the capacity of the ROB.

Note that the length of the sampling data is not confined to eight bytes in our implementation. The sampling length can be increased by using multiple 64-bit loads or 128/256-bit SIMD load operations rather than a single load (line 3 of Fig.4). For instance, we simply have 16-byte data sampling with two 64-bit loads. In this case, we use 16 F+R buffers for byte encoding.

4) EVICT+SAMPLING

The above implementation exploits an LFB as a leakage source of secret data. As the LFB holds only data that missed an L1D cache, a victim's data residing at L1D is not easily obtained by our data sampling attack. Hence, to achieve a successful attack, the implementation requires that the target data reside at levels of memory lower than L1D.

We introduce an Evict+Sampling technique to improve the sampling performance of our attack implementation. In this technique, we first evict data in the L1D cache to lower levels of memory (Evict phase). As we have no information about the location of victim's data in the L1D cache, entire lines of the cache have to be evicted in this phase. We can easily achieve this: allocate an array in memory, which is sufficiently larger than the cache (*i.e.,* more than 32 KB), and then access all the elements in the array. After eviction, we wait for a certain amount of time, during which a victim may perform load operations for secret data (Wait phase). As the data have been evicted from the L1D cache, the load misses it, which in turn causes the LFB to load the data. Finally, we execute the multibyte sampling attack to leak the data residing at the LFB (Sampling phase).

The Evict+Sampling technique is effective for an attacker who has the ability to control a victim. That is, the attacker can efficiently obtain a secret by sending a request to the victim to perform memory loads in the Wait phase. However, this technique is not only useful for these attackers. Without controlling the victim, attackers can increase the chance of leaking a secret by repeatedly executing the Evict+Sampling during the victim's execution.

```
1 void* offset = buff;
2 while (True) {
3   offset += 8;
4   if (offset >= buff_size)
5       offset = buff;
6   flush (offset);
7   memaccess (offset);
8   delayloop (cycles);
9 }
```

FIGURE 5. Code snippet of a victim application for experiments.

C. PERFORMANCE EVALUATION

We conducted experiments to evaluate the performance of our multibyte data sampling attack. In this subsection, we present the experimental results.

1) EXPERIMENTAL SETTING

For the experiments, we built three types of spy programs, MDS-64, MDS-96 and MDS-128 which perform data sampling of 8, 12 and 16 bytes per execution, respectively. The construction of all these types is based on the implementation described in Section IV-B.

We also built a program that acts as a victim, which simply continues to execute memory load operations through an infinity loop. In particular, the victim initially allocates a 64-byte buffer `buff` and fills the entire buffer with a 8-byte secret string `0xdefec7e3deadbeef`. Then, it repeatedly performs memory access to `buff` with offsets varying by eight bytes. It always flushes the offset prior to the access to ensure the string is loaded from memory via the LFB. We precisely controlled the memory loading rate of the victim by providing a certain amount of delay in the cycles between load operations. Fig.5 presents a code snippet of the victim application.

The experiments were conducted on an Intel Xeon E3-1275v6 (Kaby Lake) processor running Ubuntu 18.04 LTS 64-bit Linux. The spy and victim programs were run on the same physical core but separate logical cores by CPU pinning. While the victim program was executed, the spy performed multibyte data sampling and recorded all the obtained values to a file for subsequent processing. We emphasize that this experimental setting allows neither synchronization nor sharing of prior information by the spy and the victim.

2) THROUGHPUT AND ERROR DISTRIBUTION

We conducted an experiment to measure the throughput of each spy program for leaking secret strings from the victim. The throughput was measured while varying the number of `mul` instructions of the spy programs to determine the number of instructions that yield the best attack performance. The victim was configured to run with no delays in its loop (*i.e.,* `cycles`=0 in Fig.5) to obtain the throughput in an ideal setting. The experimental results are shown in Fig.6(a). The term 'Throughput' in the figure refers to the total bytes of secret strings successfully leaked without errors by the attack within a second.

The results show that MDS-64, MDS-96 and MDS-128 yield the best throughput with 30, 70 and 50 `mul` instructions, respectively. The measurement with fewer instructions results in lower throughput, because these instructions are not sufficient to create a large transient execution window to deliver the attack. On the other hand, more `mul` instructions degrade the throughput for all the spy programs as well. The degradation is in fact caused by the ROB capacity of the processor, which limits the total number of in-flight μ-ops of the spy program.

We also observe from the experimental results in Fig.6(a) that attacks with sampling of more than eight bytes, *e.g.,* MDS-96 and MDS-128, show a throughput of one order of magnitude less than that of MDS-64. The fact that MDS-64 results in better throughput than MDS-96 and MDS-128 is somewhat straightforward because sampling a single 64-bit word can be performed with no additional overhead in an x86-64 architecture.

Besides such an architectural limitation, we also attribute the poor throughput of MDS-96 and MDS-128 to high error rates during execution of the attacks. The errors are usually caused in the process of encoding and decoding of secret values. For instance, to handle 16 bytes of a string takes twice as long in MDS-128 as in MDS-64. As the cache state of F+R buffers is fragile and prone to corruption by noise from irrelevant system activities, it unavoidably suffers from more errors in longer encoding/decoding stages.

Fig.6(b) shows the cumulative frequency distributions of errors in leaked strings for all the spy programs. The error is represented in the graph as the normalized Hamming distance of the leaked string with respect to the secret of the victim. The distribution was obtained with the specific number of `mul` instructions that showed the best throughput for each attack. In MDS-64, more than 75% of strings leaked from the sampling attack contained no errors (*i.e.,* the Hamming distances of these strings are zero). The Hamming distance of the remaining leaked strings was more than 0.9, which indicates that they came from other memory load operations irrespective of the victim's secret string. In the cases of MDS-96 and MDS-128, the proportions of leaked strings that contained no errors were approximately 11% and 2%, respectively. As described above, relatively long stages of encoding and decoding introduce high error rates to these attacks.

3) COMPARISON TO OTHER SAMPLING TECHNIQUES

We now compare the performance of our implementation to other sampling methods. For this, we implemented another spy program that performs one-byte and 3-bytes sampling as previous work [5], [8] to leak eight bytes of the secret string from the victim.

Furthermore, we additionally implemented these sampling methods in a different way that they perform a bit-wise encoding (*i.e.,* single-bit transmission in [1]). Unlike a byte-wise

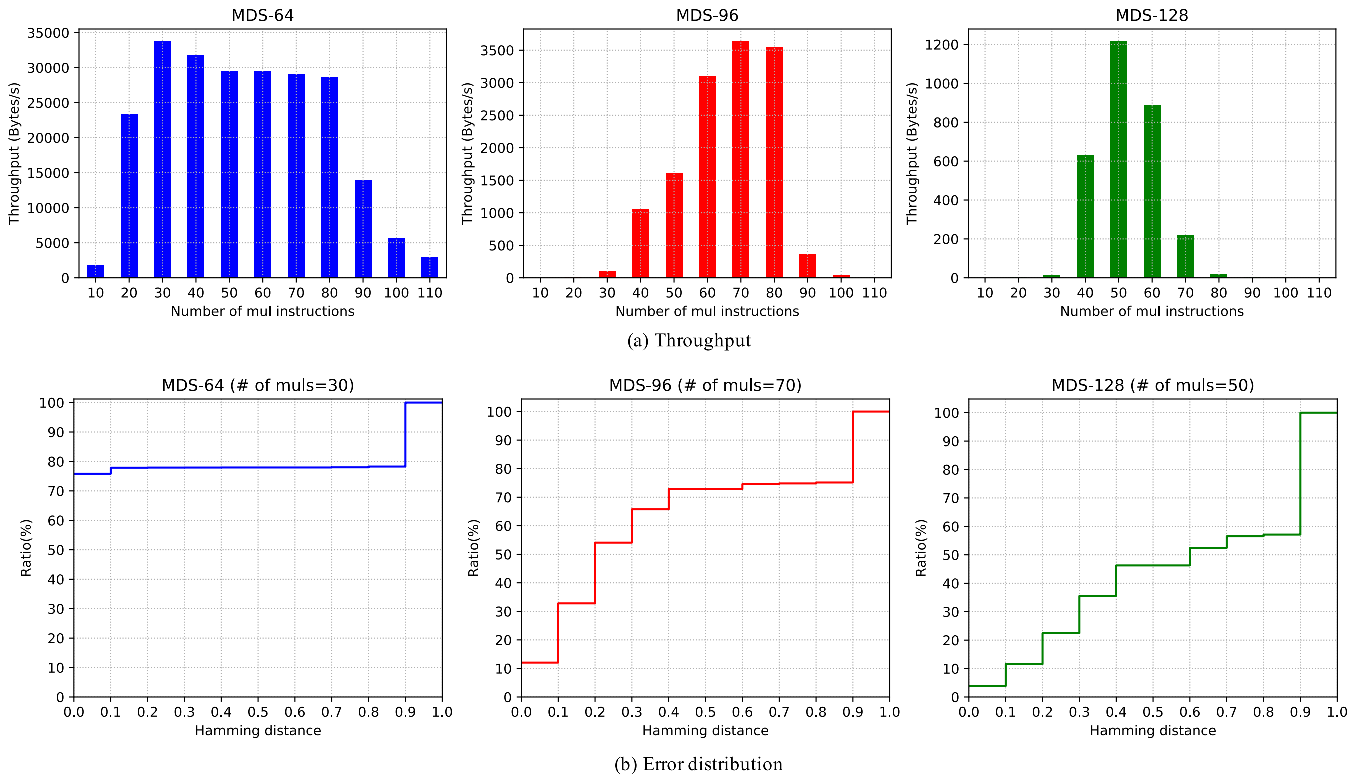


FIGURE 6. **Throughput and error distribution of the multibyte data sampling attacks.**

encoding as in our technique, each bit of the leaked secret is encoded separately to each F+R buffer in this method, where the buffer consists of only one element. Thus, the value of the bit can be decoded with only one memory access to the F+R buffer (*i.e.,* one execution of Flush+Reload measurement). This way, one-byte sampling can be performed faster as the number of memory access is reduced from 256 to 8 in the decoding stage. In contrast, however, more accesses to F+R buffers are necessary in the encoding stage. Specifically, the number of memory access is increased from one to 8 in the one-byte sampling, and from 3 to 24 in the 3-bytes sampling. Therefore, these bit-wise encoding-based sampling techniques were implemented by utilizing our window enlarging technique.

In this experiment, we used the same settings as in the previous experiment under which the spy has not been synchronized with the victim and has no prior information about the secret. Hence, we utilized only the domino-byte technique [5] among the techniques for enhancing the sampling capacity. The spy used a (4, 4)-domino byte to recover two bytes of the secret; thus four bytes for the domino bytes were additionally needed to successfully recover the entire value of the secret.

The experimental results are presented in Table 1. The term 'Executions/s' refers to the number of samplings that are executed within a second, which indicates a temporal resolution for each sampling method. The term 'Throughput (Bytes/s)' refers to the total bytes of successfully leaked secret strings without errors within a second. All the bit-wise encoding-based methods show higher temporal resolution

TABLE 1. **Performance comparison to other sampling techniques.**

	One-byte sampling		3-bytes sampling		MDS-64
Encoding method	byte-wise	bit-wise	byte-wise	bit-wise	byte-wise
Sampling capacity	1 byte	1 byte†	3 bytes	3 bytes†	8 bytes†
Executions/s	20,255	35,813	8,543	23,224	4,352
Throughput (Bytes/s)	13,496	23,762	17,845	234	34,792

†Enlarging the transient execution window is required.

than the corresponding byte-wise method owing to the fast decoding stage. Consequently, the one-byte sampling with bit-wise encoding yields the highest throughput among all the methods except MDS-64, which is also higher than that of MDS-96 and MDS-128 presented in Fig.6(a). However, the 3-bytes sampling with bit-wise encoding shows poor throughput compared to other methods. We attribute this to the encoding stage with 24 memory accesses, which is too long to be completed within a single transient window.

For MDS-64, it has the lowest temporal resolution due to the relatively long encoding and decoding stage. Despite its low speed, however, MDS-64 shows the highest throughput among all the sampling methods owing to its multibyte sampling capacity.

4) EXPERIMENTS UNDER THE SETTING OF PRACTICAL APPLICATIONS

All the previous experiments were conducted under the setting of an ideal victim application that provides the spy with the best performance: every memory load of the victim is followed by cache flushing, and the loads are repeated without delays.

TABLE 2. Success rate and throughput in various configurations (MDS-64).

Delay (cycles)	Memory loads/s	Victim(F) + Spy(NE)		Victim(NF) + Spy(NE)		Victim(NF) + Spy(E)	
		Success rate (%)	Bytes/s	Success rate (%)	Bytes/s	Success rate (%)	Bytes/s
10^4	36×10^4	0.85	22038	0.001	10	0.002	22
10^5	36×10^3	3.21	7923	0.01	5	0.36	852
10^6	36×10^2	10.58	2495	0.06	5	5.8	1356
10^7	36×10^1	5.53	93	0.86	19	4.04	81
10^8	36	7.36	13	2.47	5	4.64	7

F: Flush after memory load, NF: No flush after memory load
E: Execution with Evict+Sampling, NE: Execution without Evict+Sampling

To study the performance of multibyte sampling attacks against more practical victim applications, we conducted an additional experiment, in which various configurations were applied to the victim. First, we configured the rate of memory loads by introducing delays in the loop. As shown in Table 2, the configurations of the loop delays varied from 10^4 to 10^8 cycles, which resulted in rates varying from 36×10^4 to 36 memory loads/s. Second, we configured the behavior of cache flushing by adding a flag in the victim that controls the execution of a flush operation after loading. In addition, the spy was also configured to utilize an Evict+Sampling so that we could examine the extent of the performance gain obtained by using the technique.

Table 2 presents the experimental results of executing an MDS-64 attack against the victim under various configurations. The term 'Success rate' refers to the fraction of loads successfully leaked to the spy among all the memory loads, and 'Bytes/s' refers to the throughput of the spy.

The first combination of configurations, *i.e.,* Victim(F) + Spy(NE), achieves the best performance of all the combinations. The first combination differs from the others in the flushing behavior; that is, the victim executes without cache flushing in the other combinations. Hence, we attribute the result of the first combination to the repetitive cache flushing performed by the victim, which is ideal and hardly expected in practical scenario.

A comparison of the results of the second and third combinations shows that a significant performance improvement can be achieved against a victim without flushing, *i.e.,* Victim(NF), if the spy utilizes the Evict+Sampling technique. In particular, the spy that uses Evict+Sampling, *i.e.,* Spy(E), achieves a performance that is very close to the best performance in the situation where the victim executes memory loads at a rate less than 36×10^2 loads/s.

We conclude from the results that an MDS-64 attack with the Evict+Sampling technique is effective in leaking a secret when used against practical victim applications where the rate of secret loading is less than thousands per second.

V. EXTRACTING SESSION KEYS OF NETWORK PROTOCOLS

Secure network protocols such as TLS and SSH allow remote entities (*e.g.,* clients and servers) to communicate securely with each other over insecure networks. After a session has been established according to the protocol, all messages are exchanged between entities in encrypted form with shared session keys. As every encryption (or decryption) of messages causes session keys to be loaded from memory, secure network protocols are subject to MDS attacks.

In order to validate the effectiveness of the multibyte data sampling technique for practical applications, we demonstrate a session key extraction attack against secure network protocols. In particular, we aim to extract AES-128 and AES-256 keys, which are used in most protocols to secure their sessions. In this attack, we chose MDS-64 with the Evict+Sampling technique as our primitive, because it provides a higher throughput than the other variants.

As AES-128 and AES-256 keys are two and four times respectively larger than the length of bytes that MDS-64 can sample at a time, the implementation of an attack on AES sessions keys is not trivial. In this section, we present in detail our approach for the key extraction, including the attack model and evaluation results.

A. ATTACK MODEL

As in other microarchitectural attacks presented in the literature, in this attack we assume that both a victim and an attacker reside in the same host sharing a physical core. The victim exchanges encrypted messages with a remote entity over an established session. The session is encrypted with AES-128/256 algorithms. The attacker attempts to leak an AES session key used in the victim's session by exploiting the MDS vulnerabilities of the processor. In certain protocols such as TLS, a pair of keys are used for a session, one for the message transmission and the other for the reception. For brevity, we assume that the attacker is interested only in a reception key (*i.e.,* a decryption key), although our method presented in the next section is not confined to a single key extraction.

Regarding the attacker's ability, we suppose that no information about the victim's application is given to the attacker except information of the negotiated ciphersuite. More specifically, the attacker has no prior information of the victim, including any partial bytes of the session key or fixed strings in the victim's executable binary. In addition, we suppose that no physical memory is shared by the attacker and the victim. We emphasize that the attacker is neither the other end of the TLS/SSH communication nor performs any steps of synchronization with the victim.

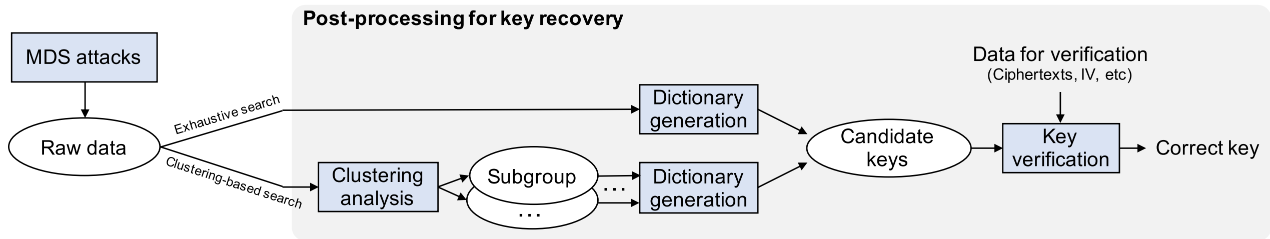


FIGURE 7. Overall process for session key extraction.

We also suppose that the attacker is able to verify whether a leaked value is a correct session key. That is, the attacker can test candidates of the session key by using public data (*e.g.,* ciphertexts and IVs) and the information about the negotiated ciphersuite.

B. SESSION KEY RECOVERY

AES Implementation: Before providing details of our session key extraction attack, we briefly describe the AES implementation of secure network protocols. Recently developed versions of secure protocols, such as TLS 1.3 and SSH-2, support AES-GCM (Galois Counter Mode) as authenticated encryption in their ciphersuite. As huge computation is necessary in such a sophisticated algorithm, most protocol applications take advantage of hardware acceleration such as AES-NI for their AES implementations. AES-NI allows to process one round of encryption with a single instruction. Each round operation is executed with two 16-byte operands, one for the round key and the other for the internal state.

The key scheduling algorithm expands session keys to a number of round keys, $R_0, R_1, \ldots, R_n$, where $n = 10$ ($n = 14$) for AES-128 (AES-256). Note that R_0 comes from the session key itself: for AES-256, both R_0 and R_1 come from the session key. These initial round keys are used to mask a plaintext block prior to subsequent round operations. All the round keys are usually precomputed at the beginning of the session and stored on the memory buffer. Whenever messages are encrypted or decrypted, the round keys are loaded from the buffer to the operands of the AES-NI instructions. All the details including the memory address of the buffer and the offsets of the round keys in the buffer depend on the AES implementations.

Key Recovery Process: For the session key extraction attack, we target the load of the initial round key (*i.e.,* R_0 for AES-128 and both R_0 and R_1 for AES-256) among all the memory loads. As the attacker has no information about the memory location of the initial round key, his/her only option is to capture the values at all the offsets accessible by MDS attacks. In particular, our MDS-64 technique can control the index offset within 64 bytes as it exploits the leakage vulnerability inside the LFB [5]. Hence, to cover all the accessible values, we execute the multibyte data sampling technique with eight different offsets at every iteration of the attack.

A session key extraction attack proceeds in two stages. In the first stage, the attacker captures all the in-flight loads at the LFB by using multibyte sampling while the victim is active in communicating with a remote entity through an encrypted session. The attacker utilizes Evict+Sampling technique to increase chances of the session key being loaded through LFB. All the leaked values are recorded in the storage together with tags indicating the offsets at which the values are leaked. Specifically, the recorded data comprise a list of tuples (v, t), where v is an eight-byte leaked value and $t(0 \leq t \leq 7)$ is the offset of v. These tuples are then used as raw data in the next stage where the postprocessing to recover a session key is performed.

Fig.7 illustrates the overall key recovery process in the postprocessing stage. The raw data obtained in the first stage are processed by applying two approaches, *exhaustive key search* and *clustering analysis-based key search*.

1) EXHAUSTIVE KEY SEARCH

As any address-agnostic data are collected by MDS attacks, the raw data obtained in the first stage contain values that are irrelevant to the secret (*i.e.,* R_0 and R_1). To determine a session key, one solution is to try exhaustively all values in the raw data to a key verification function until the correct value is determined to be successful in the verification.

Dictionary Generation: For the exhaustive key search, we first construct from the raw data a dictionary consisting of candidates for the session key. As a session key for AES-128 is 16 bytes in length, two distinct values v_i and v_j are selected from the raw data to construct a candidate key: for an AES-256 session key, four values are required. Thus, the dictionary can be generated from all possible combinations of distinct values in the raw data. As the search complexity increases together with the volume of a dictionary, we need to optimize the generated dictionary's size. To accomplish this, we exploit the tag information associated with the value. That is, only those values having continuous tags (*i.e.,* offsets) are selected for building a candidate key. By leveraging the fact that encryption keys are generally uniformly random, the dictionary size could be further minimized by filtering out unrelated data with a randomness test.

Key Verification: We verify each candidate in the dictionary by using a key verification function. The verification function is chosen according to the ciphersuite of the session. Any publicly known data including ciphertexts and IVs are used together with candidate keys for the verification.

Value in memory: `ab3c5ee3891f0fae`

Read by MDS: `a03c5ee38c1f0fae`
`ab3cdee3891f0fae`
`ab3cfee3891f0fae`
P1 P2 P3

FIGURE 8. **Errors in leaked values by microarchitectural data sampling attacks.**

2) CLUSTERING ANALYSIS-BASED KEY SEARCH

As MDS attacks are susceptible to system noise, the raw data are exposed to a non-negligible amount of errors. Thus, it does not suffice to use only the exhaustive key search approach to successfully recover the session key from such noisy data. Fig.8 illustrates an exemplary case where the secret value in the memory is `abc3c5ee3891f0fae`. Three attempts to capture the value were made, all of which resulted in leakages with only one hex-digit error in each (see P1, P2 and P3 in Fig.8). Although the correct hex-digits were found in other leaked values, the exhaustive search method could not recover the secret from the noisy values.

We can handle such a case by adaptively building a dictionary that covers all the combinations of the hex-digits. In the case shown in Fig.8, the hex-digits where errors occurred are located at three positions P1, P2 and P3. We generate a dictionary from combinations of sets, $S_{P1} = \{0, b\}$, $S_{P2} = \{5, d, f\}$ and $S_{P3} = \{c, 9\}$ which consist of possible hex-digits at each position.

We denote by *subgroup* a set of leaked values in the raw data that originate in the same memory location. By generating a dictionary per subgroup, we can successfully recover the secret from noisy data.

Clustering Analysis: The problem of determining a subgroup in the raw data in the situation where any address-agnostic data are collected is challenging. Our approach is to perform a clustering analysis of the raw data. Specifically, we attempt to partition the dataset into a number of clusters according to the Hamming distance. As a result, each cluster corresponds to a subgroup, for which a dictionary is generated.

We make use of density-based spatial clustering of applications with noise (DBSCAN) algorithm [48] for the clustering analysis. This algorithm is suitable for our attack because it allows clusters to be found in the noisy raw data without prior knowledge of the number of subgroups.

C. EVALUATION

We evaluated the performance of the session key extraction attack in terms of the success probability of key recovery and the search complexity. In the evaluation experiment, the victim was an application running the TLS 1.3 and SSH-2 protocols. We chose OpenSSL 1.0.2u [49] and wolfSSH 1.4.3 [50] for the TLS and SSH executables, respectively, as both support AES-NI implementations. To evaluate the performance in practical applications, we built those executables without any changes such as adding artificial flush instructions. The experimental setting was the same as that described in Section IV-C.

The victim ran as a server while communicating with a remote client over a session established through the protocol. We chose AES128/256-GCM as the negotiated ciphersuites of the TLS session. In the case of the SSH session, however, only AES128-GCM was selected as the ciphersuite because AES256-GCM is not yet supported in wolfSSH. The experiment was conducted with two different configurations of the session by controlling the remote client. In the first configuration, the client sent messages to the server for 3 minutes at a rate of 100 msgs/s (*i.e.,* 100 decryptions/s), and in the second, the messages were sent for 5 minutes at a rate of 50 msgs/s. The raw data were collected from the victim under each configuration. To obtain averaged experimental results, we performed a total of 100 executions of the victim for each configuration.

After the raw data were obtained, we performed a post-processing analysis of the data to search the session key. In the case of a clustering-based search, the DBSCAN algorithm was used with the parameters set to $eps = 0.4$ and $minPts = 2$.

Table 3 shows the success probabilities of session key recovery under the two different configurations. $\Pr[\mathcal{E}]$ and $\Pr[\mathcal{C}]$ refer to the success probability of key recovery when the exhaustive and the clustering-based search, respectively, were used. When these search methods were applied together, the success probability (*i.e.,* $\Pr[\mathcal{E} \vee \mathcal{C}]$) reaches up to almost 99% for recovering AES-128 session keys on both TLS and SSH applications. Even in the case of AES-256, the method could successfully recover the session key with up to 98% probability.

The clustering-based search method greatly improved the success probability of session key recovery as shown in Table 3. However, this may be at the cost of search complexity due to the increase in the dictionary size. Table 4 presents the results of the complexity measurements of the cluster-based method. The DBSCAN clustering analysis of the raw data yielded hundreds of clusters, each of which consisted of three to eight elements of the leaked values on average. Each cluster generated a dictionary of approximately from 2,000 to 7,500 candidate keys. As the session key look-up should be performed on every cluster, the total number of candidate keys from all the dictionaries determines the overall search complexity. For AES-128, the clustering-based method successfully found the session key from approximately 2^{17} ~ 2^{19} candidate keys in total, whereas the AES-256 session key recovery required up to 2^{28} ~ 2^{37} candidate keys.

VI. COUNTERMEASURE

In this section, we present possible solutions for mitigating MDS attacks, including those proposed in this paper.

A. HARDWARE-BASED MITIGATION

After the vulnerabilities that enable MDS attacks in Intel processors were revealed, the processor vendor immediately

TABLE 3. Success probability of session key recovery.

		AES-128			AES-256		
		Pr[$\mathcal{E}$]	Pr[$\mathcal{C}$]	Pr[$\mathcal{E} \vee \mathcal{C}$]	Pr[$\mathcal{E}$]	Pr[$\mathcal{C}$]	Pr[$\mathcal{E} \vee \mathcal{C}$]
OpenSSL	100 msgs/s (3 min)	0.64	0.88	0.92	0.34	0.78	0.82
	50 msgs/s (5 min)	0.77	0.95	0.99	0.67	0.94	0.98
wolfSSH	100 msgs/s (3 min)	0.72	0.92	0.94	-	-	-
	50 msgs/s (5 min)	0.75	0.94	0.99	-	-	-

$\mathcal{E}$: an event in which a session key is recovered by an exhaustive key search method
$\mathcal{C}$: an event in which a session key is recovered by a clustering-based search method

TABLE 4. Search complexity of clustering-based method.

		Number of clusters	Cluster size	Dictionary size	Complexity AES-128	Complexity AES-256
OpenSSL	100 msgs/s (3 min)	354.73	3.99	2381.68	$2^{18.46}$	$2^{36.92}$
	50 msgs/s (5 min)	544.56	7.58	7487.3	$2^{18.69}$	$2^{28.38}$
wolfSSH	100 msgs/s (3 min)	259.34	3.87	1985.32	$2^{17.54}$	-
	50 msgs/s (5 min)	454.98	6.78	5983.9	$2^{18.01}$	-

responded by providing hardware fixes through microcode updates on the affected processors [4]. The mitigation has the processor clear the internal buffers, such as a load port, store buffer and LFB before switching to different security domains, *e.g.,* in transitions from a privileged mode (*i.e.,* a kernel) to a non-privileged mode [3]. An additional hardware fix, which flushes the L1D cache on context switching, is also useful for preventing a new variant of the MDS attack that exploits the LFB as a leakage source, although the fix was originally intended to mitigate L1 Terminal Fault (L1TF) vulnerability [51].

However, such hardware fixes fail to address all the possible cases of microarchitectural attacks [52]. In particular, they provide no protection in the case where the threads of both an attacker and a victim concurrently run on a physical core, which is the case of the session key extraction attack presented in Section V.

As hardware fixes via microcode updates do not eliminate the root cause of the vulnerabilities, several mitigation techniques that require redesign of CPU hardware are proposed. With additional hardware changes, those techniques fundamentally mitigate microarchitectural attacks by isolating execution results of transient instructions until they commit [53]–[55] or by prohibiting transient executions within pre-defined secure domain [56].

B. ISOLATION OF LEAKAGE SOURCE

MDS vulnerabilities originate in the sharing of internal buffers and the L1D cache by threads from different security domains. Thus, MDS attacks can be mitigated by isolating various leakage sources that are shared by threads. One possible solution is to deactivate simultaneous multi-threading (SMT), also known as Intel Hyperthreading, on the affected processors. As certain attacks such as LVI [52] are effective even with SMT disabled, this solution must be combined with the aforementioned hardware-based mitigations to cover all the attack cases [5]. Although this solution may be effective for security purposes, the system performance may be significantly affected when SMT is disabled.

An additional solution is to schedule the execution of a thread according to its security group [3]. That is, an OS schedules threads based on the security domain so that only threads from the same domain are allowed to run concurrently on the same physical core. This enables resource isolation among different domains without the SMT being disabled, however the OS scheduling algorithm must be modified.

C. ENCRYPTION OF PROTECTED DATA

The mitigation approaches presented above require the patches (*e.g.,* a microcode update and OS modification) to be applied at a low level of the vulnerable system. This may cause the entire system to suffer from a performance degradation due to the security patch.

Application-level mitigation can avoid an overall performance degradation on the underlying system. One solution is to modify applications so that all the protected data remain encrypted in the memory and decryption occurs only after the data are loaded to registers. To prevent leakage of the encryption key itself, the key is embedded in the program's executable binary (*i.e.,* instructions), and is enforced to be only loaded to specific registers when beginning the program's execution. Thus, MDS attacks leak only encrypted data and the attacker obtains no information of the data unless he/she knows the encryption key.

Palit *et al.* [57] realized the idea of protection by encryption. They proposed a compiler-assisted method that converts memory objects annotated as a secret in a source code to the objects that remain encrypted on the memory. Although their method is intended to prevent memory disclosure attacks, in our opinion it is also effective for mitigating MDS attacks.

D. ATTACK DETECTION

MDS attacks inherently induce an exceptionally large amount of cache contention on the system, because they internally make use of cache side-channel techniques to obtain the leaked data. This characteristic of these attacks regarding cache usage can be exploited to implement an intrusion detection system for MDS attacks. There are already anomaly-based detection techniques for cache side-channel attacks [20], [58]–[62]. Certain techniques are also proposed to detect a broad range of attacks including transient execution attacks by leveraging unsupervised deep learning [63] and ensemble learning [64]. All these techniques utilize a performance monitoring counter with which modern processors are equipped that provides a number of counters for various CPU events, such as cache hit or miss. We can utilize these counters to build cache-based intrusion detection systems designated not only for MDS attacks but also for any type of microarchitectural attack that includes cache contention.

VII. CONCLUSION

MDS vulnerabilities allow an attacker to obtain secret data on internal buffers through a transient execution attack. In order to leak data of arbitrary length within the buffer, previous MDS attacks relied on repetitive sampling of a few bytes due to the narrow transient execution window. However, as an MDS attacker cannot fully control the memory address, such

an approach leads to a low signal-to-noise ratio in the sampled data, which renders postprocessing extremely challenging.

In this paper, we proposed a novel multibyte sampling technique for MDS attacks. Using the proposed technique, we are able to capture multiple bytes of a secret in one execution. The implementation of multibyte sampling is not trivial, because the encoding of several bytes within a narrow transient execution window is a challenging problem. We addressed this problem by enlarging the window by exploiting the RSB-based speculation technique. We presented our implementations in detail, including MDS-64, MDS-96 and MDS-128, which respectively sample 8, 12 and 16 bytes in one execution, as well as their performance evaluation results.

In order to validate the effectiveness of the multibyte sampling technique in practice, we demonstrated AES-128 and AES-256 session key extraction attacks against TLS and SSH applications. To achieve more successful postprocessing of the sampled data, we also proposed a clustering-based search method, which allows us to efficiently recover the secret from noisy data. In our experiments, we were able to extract AES-128/256 session keys with a success probability of at least 98% and reasonable postprocessing complexity.

We also presented an overview of several possible solutions to mitigate the relevant vulnerabilities and prevent MDS attacks, including the proposed technique.

REFERENCES

[1] 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 *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 973–990.

[2] 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 *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1–19.

[3] Intel. (2020). *Deep Dive: Intel Analysis of Microarchitectural Data Sampling*. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling

[4] Intel. (2020). *Processors Affected: Microarchitectural Data Sampling*. [Online]. Available: https://software.intel.com/security-software-guidance/insights/processors-affected-microarchitectural-data-sampling

[5] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 753–768.

[6] S. van Schaik, A. Milburn, S. Osterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 88–105.

[7] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant CPUs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 769–784.

[8] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1–18.

[9] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative data leaks across cores are real," in *Proc. IEEE Symp. Secur. Privacy*, May 2021, pp. 1–16.

[10] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 2109–2122.

[11] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! Speculation attacks using the return stack buffer," in *Proc. 12th USENIX Workshop Offensive Technol. (WOOT)*, 2018, pp. 1–12.

[12] 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 *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 249–266.

[13] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2018, pp. 974–987.

[14] J. V. 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 *Proc. 27th USENIX Security Symp.*, 2018, pp. 991–1008.

[15] O. Weisse, J. V. Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," Tech. Rep., 2018. [Online]. Available: https://limo.libis.be/primo-explore/fulldisplay?docid=LIRIAS2089352&context=L&vid=Lirias&search_scope=Lirias&tab=default_tab&lang=en_US

[16] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," 2018, *arXiv:1806.07480*. [Online]. Available: http://arxiv.org/abs/1806.07480

[17] Intel, "Intel analysis of speculative execution side channels," Intel, Santa Clara, CA, USA, Tech. Rep., 2018. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf

[18] J. Horn, "Speculative execution, variant 4: Speculative store bypass," Tech. Rep., 2018. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=15282018

[19] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *Proc. Eur. Symp. Res. Comput. Secur. (ESORICS)*, in Lecture Notes in Computer Science, 2019, pp. 279–299.

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

[21] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23th USENIX Secur. Symp.*, 2014, pp. 719–732.

[22] B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache attacks enable bulk key recovery on the cloud," in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst. (CHES)*, 2016, pp. 368–388.

[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 605–622.

[24] H. Kim, H. Yoon, Y. Shin, and J. Hur, "Cache side-channel attack on mail user agent," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, Jan. 2020, pp. 236–238.

[25] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack," IACR Cryptol. ePrint Arch., Tech. Rep. 2014/140, 2014. [Online]. Available: https://eprint.iacr.org/2014/140

[26] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 131–145.

[27] D. Genkin, L. Valenta, and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 845–858.

[28] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, 2012, pp. 305–316.

[29] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-VM attack on AES," in *Research in Attacks, Intrusions and Defenses* (Lecture Notes in Computer Science), vol. 8688. 2014, pp. 299–319.

[30] B. Gulmezoglu, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross-VM cache attacks on AES," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 2, no. 3, pp. 211–222, Jul. 2016.

[31] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing—And its application to AES," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 591–604.

[32] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "PRIME+ABORT: A timer-free high-precision L3 cache attack using Intel TSX," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 51–67.

[33] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 549–564.

[34] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 897–912.

[35] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 990–1003.

[36] Y. Shin, "Cross-VM cache timing attacks on virtualized network functions," *IEICE Trans. Inf. Syst.*, vol. E102.D, no. 9, pp. 1874–1877, 2019.

[37] Y. Shin, D. Koo, and J. Hur, "Inferring firewall rules by cache side-channel analysis in network function virtualization," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, Jul. 2020, pp. 1798–1807.

[38] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 870–887.

[39] B. Gras, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 955–972.

[40] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi, "TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS P)*, Sep. 2020, pp. 1–13.

[41] A. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A false dependency attack against constant-time crypto implementations," in *Topics in Cryptology—CT-RSA*. 2018, pp. 21–44.

[42] D. Sullivan, O. Arias, T. Meade, and Y. Jin, "Microarchitectural minefields: 4K-aliasing covert channel and multi-tenant detection in iaas clouds," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–14.

[43] M. Lipp, V. Hažić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of AMD's cache way predictors," in *Proc. 15th ACM Asia Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 1–13.

[44] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-step: A practical attack framework for precise enclave execution control," in *Proc. 2nd Workshop Syst. Softw. Trusted Execution*, Oct. 2017, pp. 1–6.

[45] T. Kim and Y. Shin, "Reinforcing meltdown attack by using a return stack buffer," *IEEE Access*, vol. 7, pp. 186065–186077, 2019.

[46] T. Kim and Y. Shin, "High efficiency, low-noise meltdown attack by using a return stack buffer," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Jul. 2019, pp. 688–690.

[47] H. Wong. (2018). *The Microarchitecture Behind Meltdown*. [Online]. Available: http://blog.stuffedcow.net/2018/05/meltdown-microarchitecture/

[48] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. 2nd Int. Conf. Knowl. Discovery Data Mining*, 1996, pp. 226–231.

[49] OpenSSL. (2020). *OpenSSL—Cryptography and SSL/TLS Toolkit*. [Online]. Available: https://www.openssl.org/

[50] wolfSSH. (2020). *WolfSSH—Lightweight SSH Library*. [Online]. Available: https://www.wolfssl.com/products/wolfssh/

[51] Intel. (2018). *Deep Dive: Intel Analysis of L1 Terminal Fault*. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault

[52] 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 *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 54–72.

[53] 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 *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6.

[54] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 428–441.

[55] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Isolating speculative data to prevent transient execution attacks," *IEEE Comput. Archit. Lett.*, vol. 18, no. 2, pp. 178–181, Jul. 2019.

[56] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "ConTExT: A generic approach for mitigating spectre," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.

[57] T. Palit, F. Monrose, and M. Polychronakis, "Mitigating data leakage by protecting memory-resident sensitive data," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, Dec. 2019, pp. 598–611.

[58] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Appl. Soft Comput.*, vol. 49, pp. 1162–1174, Dec. 2016.

[59] M. Payer, "HexPADS: A platform to detect stealth attacks," in *Engineering Secure Software and Systems—ESSoS 2016* (Lecture Notes in Computer Science), vol. 9639. 2016, pp. 138–154.

[60] T. Zhang, Y. Zhang, and R. B. Lee, "CloudRadar: A real-time side-channel attack detection system in clouds," in *Research in Attacks, Intrusions, and Defenses*, vol. 9854. 2016, pp. 118–140.

[61] J. Cho, T. Kim, T. Kim, and Y. Shin, "Real-time detection on cache side channel attacks using performance counter monitor," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2019, pp. 175–177.

[62] J. Cho, T. Kim, S. Kim, M. Im, T. Kim, and Y. Shin, "Real-time detection for cache side channel attack using performance counter monitor," *Appl. Sci.*, vol. 10, no. 3, pp. 1–14, 2020.

[63] B. Gulmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar, "FortuneTeller: Predicting microarchitectural attacks via unsupervised deep learning," 2019, *arXiv:1907.03651*. [Online]. Available: http://arxiv.org/abs/1907.03651

[64] M. Mushtaq, J. Bricq, M. K. Bhatti, A. Akram, V. Lapotre, G. Gogniat, and P. Benoit, "WHISPER: A tool for run-time detection of side-channel attacks," *IEEE Access*, vol. 8, pp. 83871–83900, 2020.

YOUNGJOO SHIN received the B.S. degree in computer science and engineering from Korea University, Seoul, South Korea, in 2006, and the M.S. and Ph.D. degrees in computer science from KAIST, Daejeon, South Korea, in 2008 and 2014, respectively. From 2008 to 2017, he was with the National Security Research Institute (NSR), Daejeon, as a Senior Researcher. From 2017 to 2020, he was with Kwangwoon University, Seoul, as an Assistant Professor. He is currently an Assistant Professor with the School of Cybersecurity, Korea University. His research interests include system and network security, CPU micro-architectural security, cloud computing security, and vulnerability analysis on embedded systems.