MDPeek: Breaking Balanced Branches in SGX with Memory Disambiguation Unit Side Channels

Chang Liu
Tsinghua University
Beijing, China
cliu21@mails.tsinghua.edu.cn

Shuaihu Feng
Zhongguancun Laboratory
Beijing, China
f3ng5h@gmail.com

Yuan Li*
Zhongguancun Laboratory
Beijing, China
lydorazoe@gmail.com

Dongsheng Wang
Tsinghua University
Beijing, China
wds@tsinghua.edu.cn

Wenjian He
Huawei Technologies Co., Ltd.
Shanghai, China
wheac@connect.ust.hk

Yongqiang Lyu
Tsinghua University
Beijing, China
luyq@tsinghua.edu.cn

Trevor E. Carlson
National University of Singapore
Singapore
tcarlson@comp.nus.edu.sg

Abstract

In recent years, control flow attacks targeting Intel SGX have attracted significant attention from the security community due to their potent capacity for information leakage. Although numerous software-based defenses have been developed to counter these attacks, many remain inadequate in fully addressing other, yet-to-be-discovered side channels.

In this paper, we introduce MDPeek, a novel control flow attack targeting secret-dependent branches in SGX. To circumvent existing defenses, such as microarchitectural state flushing and branch balancing, we exploit a new leakage source, the Memory Disambiguation Unit (MDU). We present the first comprehensive reverse engineering on the MDU's enable and update logic. Based on our detailed analysis, we develop a methodology to identify vulnerable workloads in real-world applications. We demonstrate the effectiveness of MDPeek with end-to-end attacks on the latest versions of three SGX-secured applications, including Libjpeg, MbedTLS and WolfSSL. In addition, we propose a low-overhead mitigation technique, store-to-load coupling, which provides a 7× latency reduction compared to naive techniques like serialization and load aligning.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**; **Hardware reverse engineering**; • **Computer systems organization** → *Architectures.*

Keywords: side-channel attack; memory disambiguation unit; Intel SGX; hardware security

*Corresponding author

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/2025/03
https://doi.org/10.1145/3676641.3716004

ACM Reference Format:
Chang Liu, Shuaihu Feng, Yuan Li, Dongsheng Wang, Wenjian He, Yongqiang Lyu, and Trevor E. Carlson. 2025. MDPeek: Breaking Balanced Branches in SGX with Memory Disambiguation Unit Side Channels. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3676641.3716004

1 Introduction

Intel SGX is designed to protect the code and data of user applications from being accessed by other programs or even untrusted operating systems. Due to the robust security guarantees it offers, SGX has been widely adopted in various security-critical applications [30, 36, 43]. However, since SGX's original design doesn't protect against side-channel attacks [49], a number of these attacks have been developed [22, 28, 40, 41, 44, 50, 55, 57, 63, 65], posing a significant threat to the security of SGX-based applications. Among these, control flow attacks on SGX have drawn substantial attention from researchers due to their ability to extract instruction-level information.

In response to these growing threats, Intel has recently introduced AEX-Notify [10], a mechanism to monitor the interrupts of SGX applications, thereby preventing attackers from repeatedly hijacking the CPU to perform side-channel attacks. However, AEX-Notify requires microcode updates to enable new instruction features, limiting its applicability to modern servers. For devices with older CPU versions, Intel shifts the burden to software developers [23], requiring them to build and validate side-channel-resistant applications.

Currently, software-based defenses against control flow attacks in SGX primarily fall into three categories. First, data-oblivious programming eliminates secret-dependent control flow in applications, effectively blocking all control flow attacks [46, 60]. However, it imposes a substantial performance overhead [44, 57]. Second, for microarchitecture side channels with known leakage sources, such as branch predictors [65, 66] and hardware prefetchers [9], flushing

```c
// if u >= v
if (mp_cmp(&u, &v) != MP_LT) {
    // u = u - v
    mp_sub(&u, &v, &u);
}
else {
    // v = v - u
    mp_sub(&v, &u, &v);
}
```

(a) mp_invmod_slow in WolfSSL v5.7.2

```c
// if u >= v
if (cmp_mpi(&u, &v) >= 0) {
    // u = |u - v| / 2
    sub_abs(&u, &u, &v)
    shift_r(&u, 1)
}
else {
    // v = |u - v| / 2
    sub_abs(&v, &v, &u)
    shift_r(&v, 1)
}
```

(b) mbedtls_mpi_gcd in MbedTLS v3.6.1

```c
// if A >= N
if (cmp_abs(A, N) >= 0) {
    // N->p[0:n] -= A->p[0:n]
    sub_hlp(n, N->p, A->p)
}
else {
    // dummy
    // for branch balancing
    sub_hlp(n, A->p, T->p)
}
```

(c) mpi_montmul in MbedTLS v2.6.1

```c
// if 8*8 pixels are the same
if (inptr[8] == 0 && inptr[16]
    == 0 && inptr[24] == 0 &&
    inptr[32] == 0 && inptr[40]
    == 0 && inptr[48] == 0 &&
    inptr[56] == 0) {
    // simpler logic
}
else {
    // more complicated logic
}
```

(d) jpeg_idct_islow in Libjpeg v9f

Figure 1. Secret-dependent branches in cryptographic libraries [37, 38, 62] and non-cryptographic libraries [29]. In this research,we demonstrate attacks against the targets present in (a) and (d).

the microarchitectural state of these units during context switches, as mentioned in previous research [9, 65], prevents these hardware units from leaking information about the victim's execution. Third, for unknown or complex side channels, such as interrupt-based side channels [41, 44, 63], reducing observable differences can prevent attackers from inferring the victim's control flow. For instance, in the case of secret-dependent branches, balancing both branch directions (i.e., ensuring instruction counts and types are identical on two branch paths) helps to improve control flow indistinguishability. Both flushing known leakage sources and balancing branches, up to now, have been both practical and effective in mitigating the control flow attacks on SGX.

Despite these defenses, we aim to show that these vulnerabilities persist even with state-flushing and branch-balancing techniques. In other words, there are still uncovered attack vectors that can bypass existing defenses and threaten SGX security. To demonstrate this, we present MDPeek, a novel side-channel attack that leverages a new leakage source, the Memory Disambiguation Unit (MDU). The MDU, a recently studied hardware unit in Intel CPUs, predicts whether a load can bypass an earlier store to execute out of order [12]. Although the MDU's design has been partially characterized [45], it has not been exploited as a leakage source in side-channel attacks. This is primarily due to the fact that several features, such as the conditions for an MDU update to occur, remain unexplored. Consequently, real-world code that results in an MDU update has not been straight-forward.

In this study, we first perform an in-depth reverse engineering of the MDU's enable and update logic. The enable logic governs when the MDU can trigger predictions or updates, while the update logic dictates how the MDU updates its internal state. Given that the MDU update relies on stores with delayed address generation, we also investigate the capability of various instructions to delay their target operands. Building upon these insights, we establish a set of exploitable code patterns for the MDU side channel and, using our automated analysis methodology, successfully identify multiple exploitable code snippets in the real-world applications.

Next, we develop an efficient and reliable MDU probing primitive and implement the first MDU-based side-channel attack, MDPeek. To demonstrate the practicality of MDPeek, we carry out end-to-end attacks on the latest versions of three SGX-secured applications, successfully leaking RSA keys from WolfSSL [62], RSA-CRT keys from MbedTLS [38], and images processed by Libjpeg [29]. Finally,we design a software-based mitigation technique called store-to-load coupling to prevent MDU updates and effectively defend against MDPeek, which incurs a performance overhead of less than 20%, and is significantly lower than other naive defenses, such as instruction serialization and branch balancing on load PC addresses (resulting in an overhead of up to 140%).

Contributions. The contributions are as follows:

- We propose a novel MDU-based side-channel attack, MDPeek, capable of bypassing existing defenses such as branch balancing. To the best of our knowledge, this is the first side-channel attack exploiting the MDU.
- We perform an in-depth analysis of the MDU's update mechanism at both the microarchitecture and ISA levels, developing exploitable code patterns and identifying vulnerable loads in real-world applications.
- We successfully implement end-to-end attacks on three real-world applications, achieving a high success rate with only a single trace.
- We design an efficient defense against MDPeek, called store-to-load coupling, achieving a performance improvement of approximately 7 times than other defenses.

Disclosure. We followed a 90-day disclosure period and reported the MDU vulnerability to Intel, and it has been confirmed. Intel has emphasized the critical importance of implementing side-channel-resistant codes.

2 Background and Related Work

2.1 Secret-dependent Branches

Secret-dependent branches are a common target in control flow attacks, where the branch target is determined by a secret value. In cryptographic libraries, such branches are often employed in the implementation of integer algorithms, including modular multiplication [37], greatest common divisor (GCD) [38], and modular inverse [38, 62]. In non-cryptographic libraries, these branches are often used to select different execution paths based on user input [9] or to optimize performance in specific edge cases [29].

Figure 1 illustrates four code snippets containing secret-dependent branches. In the binary extension Euclidean algorithm (BEEA) (a) [62] and the Euclidean algorithm (b) [38], the branches update the larger value during each iteration. In the Montgomery multiplication algorithm (c) [37], the branch performs a modular operation. In the inverse discrete cosine transform (IDCT) algorithm (d) [29], the branch handles edge cases where a block of pixels in the input image is identical. These algorithms are widely used across various applications. For example, BEEA and Euclidean algorithms are used in RSA and RSA-CRT, Montgomery multiplication is used in ECDSA, and IDCT is used in image decoding.

2.2 Control Flow Attacks in SGX and Their Defenses

The behavior of SGX applications can influence CPU architecture (e.g., interrupts) and microarchitectural states (e.g., cache [19, 64] and branch prediction units [14, 28, 66]), potentially leaking sensitive information, such as the target of secret-dependent branches. Current attacks on secret-dependent branches can achieve instruction-level granularity, allowing attackers to precisely determine which instructions are executed along the branch path.

Table 1 provides a summary of various branch attacks targeting SGX. These attacks can be broadly categorized into architecture-level and microarchitecture-level attacks. Architecture-level attacks primarily exploit page faults to trigger CPU interrupts, leaking control flow information through the order of page faults [20, 58, 63], the timing of fault handling [44, 57], or the number of instructions between page faults [41]. Microarchitecture-level attacks, on the other hand, infer which instructions have been executed by observing updates to the state of microarchitectural components. Well-studied components, such as cache [17, 40], TLB [18, 59], BPU [28, 65, 66], execution port [2], and prefetcher [9], have been well studied and exploited to develop powerful attacks.

To defend against these attacks, several feasible defenses have been proposed by the authors of the aforementioned works. Among these, eliminating the dependence between secrets and branches through data-oblivious programming is the most fundamental solution, as it can prevent all such attacks. However, this approach is algorithm-specific [16] and inevitably introduces performance overhead [44] (e.g., as high as 24× as reported in [11]), making it outside the scope of this research.

Beyond this, two more general and practical defense techniques have been developed. First, for attacks with known leakage sources, attackers can be prevented from observing changes in the state of microarchitectural units by flushing components during SGX context switches [9, 65].

Second, for architecture-level attacks or those involving unknown leakage sources, branch balancing can be employed to make branch paths hard to distinguish for attackers. For example, both paths can access the same pages in the same sequence to mitigate page table attacks [58, 63], or the program counter (PC) of stores on both paths can be aligned to 16 bytes from the branch to mitigate frontal attacks [44].

These two defenses are entirely software-based, making them effective in blocking the attacks listed in Table 1 across different hardware environments. However, as demonstrated in this study, these defenses only address disclosed attacks or known leakage sources. For other, as yet uncovered or unstudied microarchitectural units, it remains possible to bypass these defenses and carry out control flow attacks.

2.3 Overview of the Memory Disambiguation Unit

The memory disambiguation unit (MDU) is a hardware unit recently discovered in the 6th generation and later Intel CPUs [45], which is used to predict whether a load can be executed out of order when its data dependence with a preceding store is unknown, thereby improving the performance of the load [12]. A similar unit also exists in ARM CPUs [33], though their design details are different. According to a previous research [45], as shown in Figure 2, the MDU in Intel CPUs consists of 256 counters, selected by the lowest 8 bits of the load's PC. Each counter has 4 bits, and when a counter is cleared, the MDU predicts that a load can be executed out of order. Otherwise, the load is predicted to be blocked until its data dependence with all preceding stores is resolved.

The functionality of the MDU is illustrated in Figure 2. When a store's address generation is delayed, the CPU is unable to determine the data dependence between the store and a younger load until the store's address is generated. In such cases, the MDU is enabled (❶), and the CPU selects an MDU counter to make a prediction based on the lowest 8 bits of the load's PC (❷). The MDU predicts whether to execute the load out-of-order based on the counter (❸). After the actual data dependence between the store and load is resolved, and then in a particular pipeline stage, which have not been studied before our research, the CPU corrects any mispredictions (e.g., by rolling back and re-executing the load if it is incorrectly executed out of order) and updates the MDU counter (❹). If the store and load are aliased (i.e., a data dependence exists between the store and load), the counter is set to 15. Otherwise, the counter is decreased by 1 until it reaches 0. The load is predicted to execute out of order when the counter is 0. Otherwise, the load is blocked.

Table 1. Overview of branch attacks on SGX and suggested defenses. We do not include data-oblivious programming in defenses because it is algorithm-specific and is hard to apply on non-cryptographic libraries, according to [16].

Attack Type / Name	Attack Feature				Suggested and Currently Available Defenses*
	Leakage Source	Granularity	Distinguishable Difference	Instruction Required	
Page Table [20, 58, 59, 63]	—	4 KiB	Page	—	Branch Balancing (page)
Nemisis [57]	—	1 B	Instruction Type	—	Branch Balancing (instruction type)
Copycat [41]	—	1 B	Instruction Count	—	Branch Balancing (count between pages)
Frontal Attack [44]	—	1 B	Store PC	Store	Branch Balancing (store 16-byte aligned)
TLBleed [18]	TLB	4 KiB	Page	—	Disable SMT
Branch Shadowing [28]	LBR	1 B	Instruction PC	Branch	Branch Obfuscating (Zigzagger)
Bluethunder [22]	PHT, PHR	1 B	Instruction PC	Branch	PHT Flushing
NightVision [66]	BTB	1 B	Instruction PC	—	BTB Flushing
PathFinder [65]	PHT, PHR	1 B	Instruction PC	Branch	PHT and PHR Flushing
PortSmath [2]	Port	1 B	Instruction Type (uOP)	—	Disable SMT
Cache [6, 17, 21, 40, 51]	Cache	64 B	Cache Line	—	Cache Flushing
AfterImage [9]	Prefetcher	1 B	Load PC	Load	Prefetcher Flushing
MDPeek (This Work)	**MDU**	**1 B**	**Load PC**	**Load**	**Bypassing All Defenses Above**

* We do not include defenses requiring hardware modification because they may not be available currently.

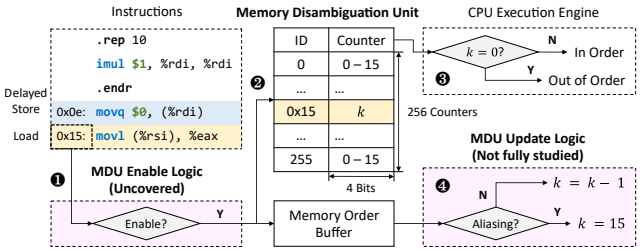

Figure 2. The structure and functionality of the MDU. Some parts still remain unknown before this research.

In prior research [45], incorrect MDU predictions are exploited to construct a new variant of the Spectre-V4 attack [7]. In this attack, the attacker uses carefully designed code (e.g., as shown in Figure 2) to train the MDU and enable it, and subsequently injects a gadget [27] into the victim's execution to trigger speculative execution under incorrect predictions. However, previous work does not delve into the enable and update mechanisms of the MDU in detail. As a result, two critical questions remained unresolved: (1) Under what conditions is the MDU enabled? (2) Which code snippets in real-world applications can trigger MDU updates?

3 Overview of MDPeek

3.1 Motivation

In this work, we aim to demonstrate that, while the software defenses described in Table 1 can defend against some of these attacks, the balanced branches in SGX are still vulnerable. We select the MDU as a new leakage source to bypass flushing-based and branch balancing defenses. We hope that this work encourages researchers to reassess the effectiveness of balanced branches and consider more robust and general defenses to protect SGX applications that still contain secret-dependent branches.

3.2 Threat Model

Consistent with SGX's security model and similar to previous work, we assume the attacker controls an untrusted OS with root privileges. The attacker can manage page tables, handle interrupts, and schedule processes. The attacker aims to probe balanced and secret-dependent branches in victim applications through a single trace. The attacker has prior access to the victim's code, which contains secret-dependent branches and load instructions along branch paths that can update the MDU. During the attack, both the attacker and the victim execute their processes on the same physical CPU, allowing them to share the MDU.

The victim is an application running inside SGX, containing secret-dependent branches. The developers are aware of side-channel threats and have implemented the defenses outlined in Table 1. Specifically, the victim disables SMT, employs eviction sets to flush the cache, prefetcher, and BPU states during context switches, and implements branch balancing as described in Table 1 using LLVM [35]. This ensures both branch sides execute the same instruction types [57], maintain the same instruction count between pages [41], and align stores' PC to 16 bytes from the branch [44].

3.3 Workflow

The attack workflow of MDPeek is illustrated in Figure 3. First, the attacker manipulates the page table to trigger a

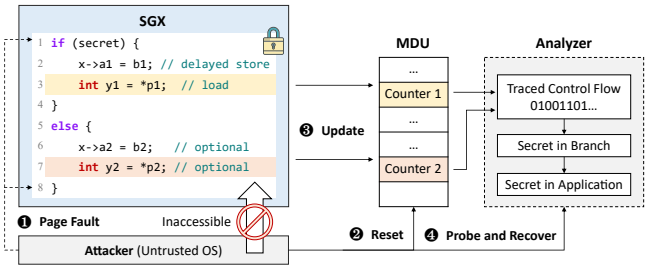

Figure 3. Workflow of MDPeek. The attacker holds the control of an untrusted OS, and the victim is an application running in SGX.

page fault both before and after the balanced branch (❶). At the first page fault, the attacker initializes the MDU counters in the interrupt handler using code from their own address space (❷). The attacker then yields the CPU to the victim, allowing the branch to execute. Along different branch paths, at least one load updates an MDU counter (❸). After the branch is executed, the attacker regains control of the CPU through another page fault, uses the probing primitive to detect changes in the MDU counters, infers the branch path, and recovers the application's secret based on the information leaked via MDPeek (❹).

To implement MDPeek, it is essential to identify the code responsible for updating the MDU. In Section 4, we reverse-engineer the MDU's enable and update logic from the microarchitecture perspective. In Section 5, we develop a code pattern for MDU updates and create an automated tool to search for exploitable code snippets in real-world applications. Furthermore, to achieve a high-accuracy, single-trace attack applicable in scenarios such as RSA key generation, we design a method for automatically generating efficient attack primitive, discussed in Section 6.1.

4 Reverse-Engineering the MDU Enable and Update Logic

In this section, we perform an in-depth reverse engineering of the MDU's enable and update logic. The enable logic governs when the MDU can trigger predictions, while the update logic dictates how the MDU updates its internal state.

4.1 Method

For an overview of the reverse engineering of the MDU, we use the microbenchmark shown in Figure 2 to deterministically update the MDU. First, we execute 15 non-aliased store-load pairs (i.e., store and load instructions with 2 different addresses) to clear a MDU counter. Next, we run an aliased store-load pair to increase it to 15. Following this setup, we execute a test case containing non-aliased store-load pairs 15 times. To ensure that the loads in the test case and the microbenchmark select the same MDU counter, we insert nop instructions before the function entry of the test case, aligning the lowest 8 bits of the load's PC.

After executing the test case, we rerun the microbenchmark with an aliased store-load pair. If the test case updates the MDU, the counter will be updated to 0, causing the MDU to incorrectly predict that the load will execute out of order, triggering a misprediction. The CPU will then roll back, squash the load from the reorder buffer (ROB), and re-execute the load. Conversely, if the test case does not update the MDU, the load will be predicted to execute in order. We use the Performance Monitor Counter (PMC) event `MACHINE_CLEARS.COUNT` [24] to detect if a CPU rollback occurs, allowing us to infer whether the MDU is updated.

For each test, we bind the test case and microbenchmark to the same CPU core and repeat the execution 10,000 times to mitigate noise. To prevent the compiler from introducing unintended stores and loads, we write the test case directly in assembly. We conduct experiments across 8 different Intel microarchitectures listed in Section 6 and observe similar designs. In this section, we present the results from our experiments on the Intel Core i7-6700K CPU (Skylake) [4].

4.2 Effects of Delay Type

After an instruction is dispatched from the Instruction Decode Queue (IDQ) to the backend, it passes through several pipeline stages. For example, a store needs to generate the data and its address, translate the address, and then send the request to the cache or memory. If no exceptions occur, the store proceeds to commit and retire. Each stage can introduce delay, potentially slowing down the store. To study the impact of delays at different stages of store and load on the MDU, we design the experiment shown in Figure 4.

We induce delays in the generation of store and load addresses by introducing port contention [2]. We trigger TLB misses using PTEditor [39] to delay address translation for both the store and load. Additionally, we flush the cache to introduce execution delays, and insert slow instructions, such as `movntdqa` [31], before the store and load to delay their retirement. The results show that the enabling and update of the MDU depend on two key conditions: (1) the store address must not yet be generated when it enters the CPU backend; and (2) the store address generation must take longer than the load. We also find that delays at other stages do not affect the enabling or update of the MDU.

4.3 Effects of Unresolved Data Dependence

According to the design motivation of the MDU, predictions are necessary only when the data dependence between the store and load is uncertain. To verify this, we conduct the experiment shown in Figure 5. We delay the address generation of the store (i.e., the generation of `rdi`) using a preceding load, and then observe whether a younger load (i.e., the load accessing the address `rsi`) updates the MDU. When no other instructions are present between the store and load, the CPU

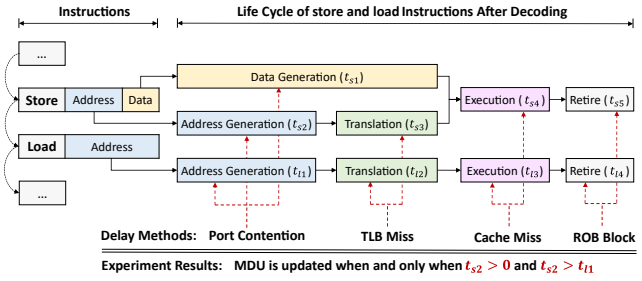

Figure 4. Experiments on the effects of the delay type on the MDU update, which shows that only the delay of store's address generation enables the MDU.

```
mdu_update_dependence_1:
    movq (%rdi), %rdi
    movq $0, (%rdi)
    nop
    nop
    nop
    movq (%rsi), %rsi
    lfence
    ret
```
Update rate: 100%

```
mdu_update_dependence_2:
    movq (%rdi), %rdi
    movq $0, (%rdi)
    mov %rdi, %rax
    and $0, %rax
    or %rax, %rsi
    movq (%rsi), %rsi
    lfence
    ret
```
Update rate: 0

```
mdu_update_dependence_3:
    movq (%rdi), %rdi
    movq $0, (%rdi)
    mov %rdi, %rax
    movq $0, %rax
    or %rax, %rsi
    movq (%rsi), %rsi
    lfence
    ret
```
Update rate: 100%

Figure 5. Experiment codes which prove unresolved data dependence is necessary to update the MDU. The codes are written in AT&T assembly code format.

is unable to determine whether `rdi` and `rsi` are equal before `rdi` is generated, thus it cannot determine the dependence between the store and load, leading to an MDU update.

Next, we introduce arithmetic instructions before the store and load, ensuring that they do not alter the semantics of the test case. For example, we introduce an additional register `rax` and establish an explicit data dependence through a bitwise operation: `rsi = (rdi & 0) | rsi`. We observe that the MDU does not update because the data dependence between the store and load is now deterministic. The resolved dependence may prevent the MDU from being updated in two ways. First, the CPU identifies the dependence of the store and load as soon as they are issued (`rdi` is computed from `rsi`), and directly disables the MDU. Second, it delays the generation of the load address, causing the update condition described in Section 4.2 to remain unsatisfied.

Finally, we replace the `and` instruction with a `mov` (or alternatively, use `xor rax, rax`), eliminating the data dependence between `rax` and `rdi`. After moving 0, `rax` no longer acts as an intermediary register between `rdi` and `rsi`. As a result, the MDU now updates its internal state, indicating that it has observed an unresolved data dependence. This, in addition to the delayed store address generation, is a necessary condition for enabling and updating the MDU.

4.4 Effects of ROB on the MDU

After determining the necessary conditions for MDU updates, we aim to investigate how many instructions can be placed

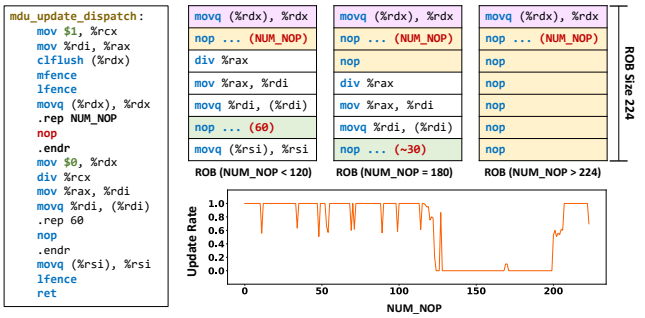

Figure 6. Experiment code and result which prove the delayed store and load should appear in the ROB simultaneously, so that the load can enable the MDU.

```
mdu_update_commit_squash:
    movq $0, (%rdx)
    clflush (%rdx)
    mfence
    lfence
    movq (%rdx), %rdx
    movq (%rdx), %rdx
    .rep 10
    imul $1, %rdi
    .endr
    movq %rdi, (%rdi)
    movq (%rsi), %rsi
    lfence
    ret
```

	Store Address (%rdi)	Load Address (%rsi)	MDU Update
Exp. 1	Valid, TLB Hit	Valid, TLB Hit	✓
Exp. 2	Valid, TLB Miss	Valid, TLB Hit	✓
Exp. 3	Valid, TLB Hit	Valid, TLB Miss	✗
Exp. 4	Valid	Invalid	✗
Exp. 5	Invalid, not generated	Valid	✗
Exp. 6	Invalid, page offset = 0x0	Valid	✓
Exp. 7	Invalid, page offset = 0x0	Valid, page offset = 0x4	✗
Exp. 8	Invalid, page offset = 0x0	Valid, page offset = 0x5	✓

Figure 7. Experiment code and result which prove an uncommitted load with a valid physical address can still update the MDU.

between a delayed store and a load while still allowing the MDU to work and update. To explore this, we conduct the experiment illustrated in Figure 6. At the top of the ROB, we insert a load and delay its commit time by inducing a cache miss. Before this load commits, subsequent instructions, including `nop`s, occupy ROB entries but cannot commit. Next, we use a `div` instruction to delay the address generation of a store. The `div` introduces a substantial delay to store address generation, allowing us to insert up to 60 `nop`s between the store and load while ensuring that the store address has not yet been generated when the load is issued. Prior to the `div`, we insert `NUM_NOP` `nop`s to control whether the store and load can reside in the ROB simultaneously.

For instance, when `NUM_NOP` is small, both the store and load can enter the ROB before the first load commits. When `NUM_NOP` reaches around 180, the store and load cannot reside in the ROB at the same time due to the ROB size limitation of 224 on Skylake CPU. As shown in Figure 6, we observe that the MDU updates only when both the store and load are present in the ROB simultaneously. Therefore, the number of instructions (more specifically, μops) between the store and load must not exceed the ROB size.

4.5 Effects of Commit and Squash Stages on the MDU

To further analyze the complete MDU update logic, we investigate when and how a load updates the MDU in this section. To determine whether the load updates the MDU during the

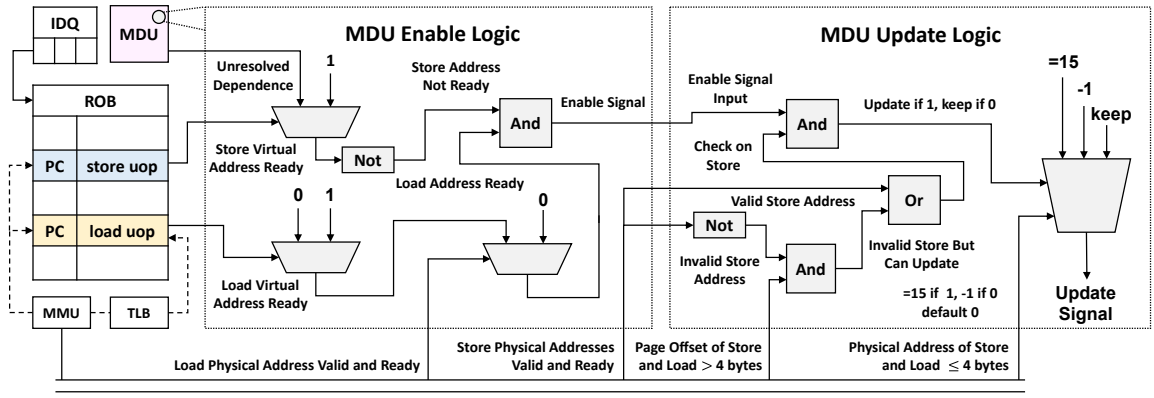

Figure 8. Enable logic and update logic in the MDU reverse-engineered in this paper. The MDU is enabled when only when a store's address is delayed and CPU cannot determine the dependence of the store and a younger load. The MDU is updated only when it is enabled and the load's physical address is valid and ready.

commit stage, we design the experiments (Exp. 1 to Exp. 8) shown in Figure 7. By accessing an unmapped address, we trigger a page fault. Similar to the Meltdown attack [32], we use an interrupt handler to capture the exception and then probe whether the uncommitted load updates the MDU. The results from Exp. 1 show that the load can still update the MDU, indicating that MDU updates occur before commit.

Using PTEditor [39], we modify the address attributes of the uncommitted store and load to further study when the MDU updates. Exp. 2 and Exp. 3 show that if the load encounters a TLB miss, the MDU does not update. Thus, MDU updates occur after the load's address translation but do not depend on the store's physical address. Exp. 4 indicates that, for the MDU to update, the load address must be valid and have a physical address mapping. Exp. 5 and Exp. 6 demonstrate that while the store's address validity is not required, the virtual address must be generated for the MDU to update. Exp. 7 and Exp. 8 reveal that even if the store has an invalid address, the MDU can still update as long as the page offsets of the store and load are larger than 4 bytes.

These experiments provide valuable insights for the design of MDPeek. For instance, when implementing MDPeek, interrupts used for synchronization do not interfere with the load's ability to update the MDU, as the MDU can still be updated due to out-of-order execution even after an interrupt occurs. Additionally, the experiments indicate that for the load to update the MDU, it must result in a TLB hit, which is typically the case in real-world applications.

4.6 Effects of Multiple Stores and Loads on the MDU

After analyzing the enable and update logic, we study the impact of multiple stores and loads on MDU updates. First, we find that when multiple stores precede a load, the MDU can be enabled and trigger predictions as long as the data dependence between one of the stores and the load is unknown. However, the update depends solely on the last store in the sequence. Specifically, only if the last store is delayed and has indeterminate data dependence with the load will the MDU update. Therefore, when searching for exploitable code, we only need to assess whether the store closest to the load meets the MDU's enable conditions.

Additionally, when a store is followed by multiple loads, the CPU evaluates each store-load pair independently to determine whether the enable conditions are satisfied and triggers separate predictions and updates for each load. Consequently, even if the CPU determines the dependence between one store-load pair, it does not affect the MDU update for other loads. This behavior suggests that multiple loads following a delayed store can all serve as exploitable targets.

4.7 MDU Reverse Engineering Summary

In this section, through a series of experiments, we comprehensively analyze the enable and update logic of the MDU, summarized in Figure 8.

Enable Logic. The CPU enables the MDU when the following four conditions are met: (1) both the store and load are present in the ROB, (2) the store's virtual address is unknown, (3) the load's address is valid and has completed address translation, and (4) the CPU cannot determine the data dependence between the store and the load.

Update Logic. Once the MDU is enabled, and assuming no additional stores are present between the store and load, the CPU evaluates whether the store meets the requirements for updating the MDU. The MDU will be updated if one of the following two conditions is satisfied: (1) the store's address is valid, or (2) the store's address is invalid, but its page offset is less than 4 bytes from the load's address. The method by which the MDU updates depends on whether the physical addresses of the store and load are identical: if they are equal, the counter associated with the load is set to 15; otherwise, the counter is decreased by one.

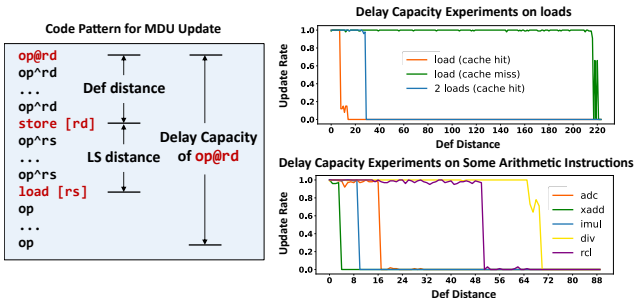

Figure 9. Code pattern for updating the MDU (left), and examples to show the effects of the delay capacity on the MDU update (right).

Compared to previous research [45], it has been shown that updating the MDU requires not only a specific store-load pair but also certain conditions related to their address generation. These findings provide valuable insights into identifying instruction sequences that can update the MDU within an application that is not under the attacker's control. As a result, our reverse engineering extends the exploitation of the MDU beyond single-process transient execution attacks to side-channel attacks against SGX. The reverse engineering in this section provides a robust foundation for identifying exploitable code in real-world applications and designing MDPeek.

5 Identify Vulnerable Loads in Applications

In Section 4, we perform a comprehensive reverse engineering on the MDU's enable and update logic. In this section, we investigate instruction sequences from the ISA perspective that can trigger the MDU's enable and update logic. The instructions we examine are part of the BASE extension set from the `uops.info` instruction database [1], which sufficiently covers most relevant attack scenarios.

The purpose of this study is to automatically identify potentially vulnerable loads in real-world applications. These loads may be capable of updating the MDU and could be exploited by an attacker. Our approach marks potentially vulnerable loads while ensuring that all unmarked loads are not exploitable, which effectively detects vulnerable loads and safely reduces overall defense overhead by skipping unmarked loads. To achieve this, we introduces the delay capacity model (Section 5.1) for evaluating address generation latency, and explains how this model is applied to identify potentially vulnerable loads (Section 5.2).

5.1 Code Pattern and Delay Capacity

Based on the results of the reverse engineering, we model the simplest code pattern that can update the MDU, with its pseudocode shown in Figure 9. Given a load instruction `load [rs]`, where the data address is `rs`, we assume the closest preceding store is `store [rd]`, with the data address `rd`. We use `op@rd` to denote the instruction preceding the store, where `rd` is the target operand, and `op^rd` represents other instructions that do not target `rd` as the operand.

We define the distance between two instructions as the number of intermediate instructions along the shortest path in the control flow graph, which is linearly and positively correlated with the execution time of an equivalent number of `nop` instructions. We refer to the distance between the store and the load in the code pattern as the *LS distance*, and the distance between `op@rd` and the store as the *Def distance*. Additionally, we define the *Delay Capacity* of `op@rd` as the maximum Def distance that results in an MDU update rate greater than 50% when the LS distance is 0. From an ISA perspective, the update condition for the load is: *Def distance* + *LS distance* < *Delay Capacity* of `op@rd`.

We develop a tool to automatically generate instructions containing a target operand from the `uops.info` database and evaluate their delay capacity. Due to space limitations, we present only a subset of these instructions in Figure 9. The delay capacity varies significantly across different instructions. For example, the delay capacity of `div` is about 60 greater than that of `imul`. Furthermore, the delay capacity of the same instruction can vary considerably depending on the microarchitectural state. For instance, if a load generates the store address and there is a cache hit, the delay capacity is only 16, meaning that the delayed store and the subsequent vulnerable load must be within 16 instructions of each other. However, if there is a cache miss, the delay capacity can approach the size of the ROB, which is 224 on Skylake CPUs.

The source operand of `op@rd` can be traced back to the destination operand of an earlier instruction, such as a sequence of `imul` instructions sharing the same destination operand. In this case, we precompute the delay capacity for a chain of identical instructions with the same destination operand, which follows a non-linear pattern. For example, the delay capacity of a single `imul` is 6, while for two consecutive `imul` instructions, it increases to 20.

For an instruction chain consisting of different instructions, we compute the delay capacity of an equivalent instruction chain for each individual instruction. Then, we take the maximum delay capacity among them as the delay capacity of the entire instruction chain. This approach ensures that we correctly identify potentially vulnerable loads.

This model represents the simplest code pattern, where the load's address `rs` is not under consideration. Nevertheless, our research demonstrates that this code pattern is sufficient to identify potentially vulnerable loads in real-world applications that are vulnerable to MDPeek. More complex code patterns could reveal even more exploitable code snippets, which we leave for future work.

5.2 Identify Exploitable Loads

In the LLVM backend, we add a function-level analysis pass using the code pattern and delay capacity information to

Table 2. Vulnerable loads we find in applications under different compilation options. Function jpeg_idct_islow has 2 secret-dependent branches, labeled as b1 and b2.

Application	Functions with Secret-dependent Branches	# of Vulnerable Loads			
		O0	O1	O2	O3
Libjpeg v9f	jpeg_idct_islow (b1)	16	1	1	1
	jpeg_idct_islow (b2)	16	8	8	8
MbedTLS v3.6.1	mbedtls_mpi_inv_mod	0	0	0	6
WolfSSL v5.7.2	mp_invmod_slow	0	0	0	18

determine whether the instruction sequence generated by LLVM [35] contains exploitable loads. For each load, we identify the closest preceding store in the same basic block or a preceding block and calculate the LS distance. Then, for that store, we calculate the Def distance between it and its define instruction to check whether the difference between the delay capacity and the Def distance exceeds the LS distance. If it does, the load is deemed exploitable.

It is important to note that the define instruction can be traced further back in the instruction stream, allowing us to continue searching for the source operands corresponding to the define instruction's operands, thereby providing more accurate delay information. Based on empirical observations, we currently use a 4-level trace-back and select the instruction with the largest difference between delay capacity and Def distance for comparison with the LS distance. The complexity of the search algorithm is $O(n^3)$, where n represents the number of basic blocks in a function. The primary bottleneck lies in calculating the distances between instructions, as determining the distance between instructions in different basic blocks requires computing the shortest path between those blocks. We use the Floyd algorithm [26] to calculate the shortest path, which has a complexity of $O(n^3)$. The shortest path may overlook additional instructions at runtime, which can prevent the MDU from being updated by some marked loads. However, this approach still achieves its goal of identifying all potentially vulnerable loads.

Using our methodology, which takes advantage of LLVM, we search for potentially exploitable code snippets in the latest versions of Libjpeg, MbedTLS, and WolfSSL. Since the focus of this paper is on bypassing existing defenses, we specifically target functions with known secret-dependent branches and use this code pattern to search for exploitable loads. Given that different optimization options produce significantly different instruction sequences, we perform searches on instruction sequences generated by four different optimization levels passed to LLVM. The results are summarized in Table 2. In Libjpeg, numerous stores and loads exist along the secret-dependent branch paths, leading to many exploitable code snippets. In contrast, developers of MbedTLS and WolfSSL are aware of the importance of

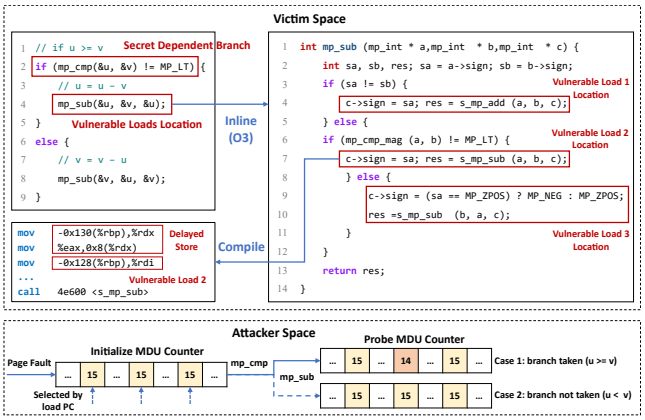

Figure 10. An example of vulnerable loads in WolfSSL and the workflow of MDPeek for probing the direction of a secret-dependent branch is as follows. When compiled with `-O3`, the function `mp_sub` within `mp_invmod_slow` is inlined. The delayed store occurs when writing to `c->sign`, while the exploitable load is used to pass a parameter to the function `s_mp_sub`. MDPeek exploits the MDU update by one of the vulnerable loads to leak whether the secret-dependent branch is taken or not.

branch balancing and have implemented balancing at the source code level, so exploitable loads are found only with the `-O3` optimization level. This is because `-O3` optimization inlines certain simple functions, exposing additional stores and loads. This phenomenon has also been exploited in previous research [44]. The exploitable loads primarily appear around function calls, where variables are fetched from the stack. A typical exploitable code snippet is shown in Figure 10. After the function `mp_sub` is inlined, its code appears along the secret-dependent branch paths. The write to the variable `c->sign` results in a delayed store, and the subsequent call to `s_mp_sub` retrieves parameters from the stack, resulting in an exploitable load.

As illusrated in Figure 10, during the MDPeek attack, the attacker hijacks control before the victim executes the secret-dependent branch and initializes the MDU counter to 15 using three aliased store-load pairs, where the load PCs are 256-byte aligned (i.e., with the same 8 least significant bits). If the branch is taken, one of the three vulnerable loads updates the MDU counter. The attacker then probes the MDU counters to infer the branch direction.

6 End-to-end Attacks with MDPeek

Since MDPeek uses a new leakage source, it is capable of bypassing existing flushing-based defenses. Moreover, MDPeek exploits the selection of the load PC's lowest 8 bits, enabling it to circumvent existing branch balancing defenses. Through our experiments, we have also demonstrated that the MDU is shared across different processes running on the same physical CPU, including processes in SGX. In this section, we implement end-to-end attacks using MDPeek on

3 real-world applications, validating the effectiveness and practicality of MDPeek.

We confirm the effectiveness of MDPeek across 8 Intel CPUs spanning different microarchitectures, including Core i7-6700K (Skylake), i7-7567U (Kaby Lake), i7-8650U (Kaby Lake R), i5-5365U (Whiskey Lake), i9-9900K (Coffee Lake), i7-10700 (Comet Lake), i7-1065G7 (Ice Lake), and Xeon E-2314 (Rocket Lake). In this section, we primarily focus on the i7-6700K CPU. We first introduce the code used to initialize and probe the MDU state, followed by the attacks on Libjpeg, MbedTLS, and WolfSSL, respectively.

6.1 The MDPeek Attack Primitive

In the attack on RSA key generation (e.g. in WolfSSL and MbedTLS), the generated key differs with each execution, requiring us to complete MDPeek in a single trace without the ability to reduce noise by repeatedly running the victim. Therefore, we need code that shows strong resistance to noise to initialize and probe the MDU state. Using a method similar to that described in Section 5.1, we automatically select data-dependent instructions and place them both before the store and after the load.

We use a non-aliased store-load pair to measure the execution time under two conditions: when the MDU counter equals 0 and when it is greater than 0, obtaining distinct timing distributions. If the counter equals 0, the MDU predicts that the load can execute out of order, leading to shorter execution times. Conversely, if the counter is greater than 0, the MDU predicts that the load will be blocked, allowing it to execute only after the store's address is generated, resulting in longer execution times. Let T_1 represent the timing distribution when the MDU equals 0, and T_2 represent the timing distribution when the MDU is greater than 0. We define an evaluation function F as follows:

$$F(T_1, T_2) = W(T_1, T_2) - 15(V(T_1) + V(T_2)) - \frac{1}{4}(M(T_1) + M(T_2)).$$

where M represents the mean of the distribution, V represents the variance, and W represents the 1-Wasserstein distance [48] between the two distributions to measure their difference. The evaluation function F selects code sequences where the distributions T_1 and T_2 are smaller, more stable, and exhibit a significant difference, making them suitable for probing the MDU state.

We automatically enumerate different instruction types and counts of data-dependent instruction sequences, and select the code with the greatest F. Based on the results, we select the `lea` instruction to introduce delays and amplify the timing difference, executing it 50 times both before the store and after the load. The probing code is shown in the snippet in Figure 11, indicating that the timing distributions for the two MDU states are stable and show approximately a 150-cycle difference, allowing for accurate probing of the MDU counter. We also use this code to initialize the MDU.

```
stld_probe:
  mov     $-8, %r9
  .rep 50
  lea     0x40(%rdi,%r9,8), %rdi
  .endr
  mov     %rdi, (%rdi)
  mov     (%rsi), %rax
  .rep 50
  lea     0x40(%rax,%r9,8), %rax
  .endr
  lfence
  ret
```

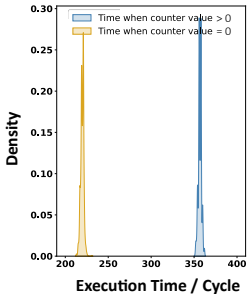

Figure 11. Code to initialize and probe the MDU counters (left), and the time distribution for two MDU states (right).

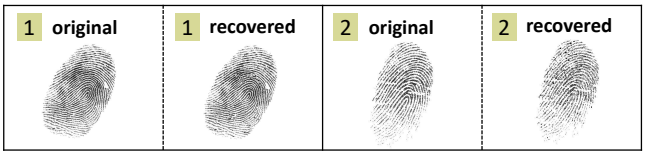

Figure 12. Example of recovered images from Libjpeg attack through MDPeek.

Specifically, we execute 3 store-load pairs with the same address to initialize a MDU counter to 15.

6.2 Attacking Libjpeg

Victim Code. During image decoding, Libjpeg uses an inverse discrete cosine transform (IDCT) to convert compressed frequency-domain data back into spatial-domain data. The algorithm processes 8x8 pixel blocks sequentially, scanning the image from the top-left to the bottom-right, and calls the `jpeg_idct_slow` function for each block, as shown in Figure 13(a). For each invocation, the algorithm executes eight iterations, each containing secret-dependent branches. In the case of a black-and-white image, the branch direction depends on the distribution of black and white pixels in the block, following a specific pattern.

Method. We synchronize with the victim code using the page fault sequence shown in Figure 13(a) and perform MDU initialization and probing. To facilitate page table manipulation, we use SGX-Step[56]. We observe that the number of taken and no-taken branches varies according to the pixel distribution pattern. For each branch, after all eight iterations, we perform a probe to record how many times the branch is taken, as shown in Figure 13(a). Finally, we use a precomputed mapping table to map the results to specific pixel distributions, in order to reconstruct the image.

Evaluation. We evaluate the effectiveness of the attack using fingerprint images from the FVC2004 dataset [15]. MDPeek takes 7.8 minutes to leak a 640×480 image on average, achieving an accuracy of 98.76% in recovering the pixels in

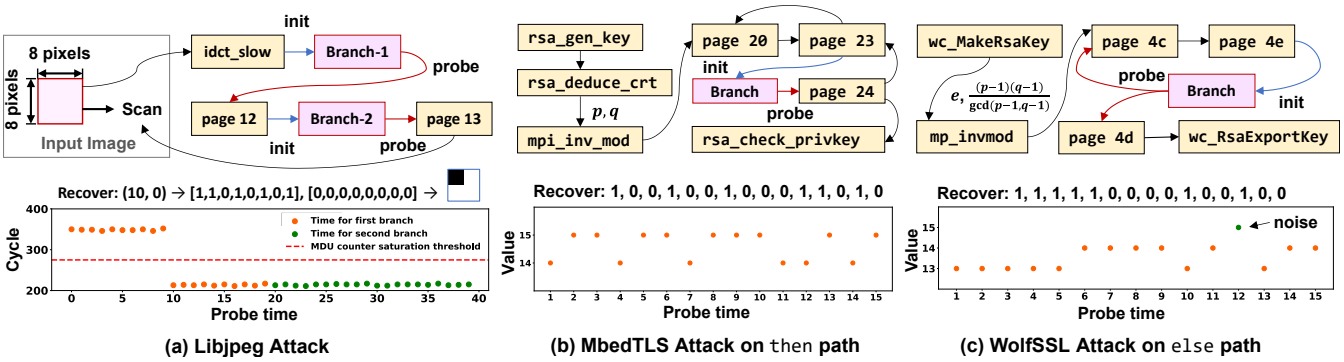

Figure 13. Examples of attack procedures and side channels for three applications. For Libjpeg (a), each secret-dependent branch is executed 8 times in each each `idct_slow` function call, and we only interrupt this function twice. For MbetTLS (b) and WolfSSL (c) attacks, we trigger a page fault of the instruction page before and after each secret-dependent branch to initilize and probe, and then infer the branch direction using the MDU side channel.

the reconstructed image that match the original image. Figure 12 shows 2 examples. Although some side-channel noise introduces small image artifacts in the form of noise dots, it does not significantly affect the overall image contours.

6.3 Attacking MbedTLS

Victim Code. When implementing the RSA-CRT key generation algorithm, MbedTLS uses the BEEA (Binary Extended Euclidean Algorithm) to compute q^{-1} mod p. The inputs to BEEA are large prime numbers p and q, and the algorithm repeatedly executes three steps: u-loop, v-loop, and sub-step [3]. The u-loop and v-loop are deterministic and can be directly obtained through the page fault sequence. The branches we attack occur in the sub-step phase, which is executed once in each iteration.

Method. We synchronize with the victim using the page fault sequence shown in Figure 13(b). Similarly, we implement it using SGX-Step. For each sub-step, there are three exploitable loads in each branch direction. We can probe any of these to recover the branch direction. We initialize the counter to 15, and if a load is executed, the counter is updated to 14, as shown in the example in Figure 13(b). Finally, based on the branch direction in the sub-step and the number of u-loop and v-loop executions in each iteration, we can calculate the iteration values for each iteration, ultimately recovering p and q. By using multiple load probe results and applying a voting mechanism, we can select the branch direction and reduce noise.

Evaluation. We use MDPeek to recover the randomly generated large primes p and q during the RSA-CRT generation of a 2048-bit key. We perform 1000 attacks with different private keys. Each attack takes about 830 ms. We probe different MDU counters selected by various loads to reduce the noise. The success rate for a single load exceeds 97%. By using two loads, the noise can be effectively mitigated, allowing the key to be recovered with nearly 100% accuracy.

6.4 Attacking WolfSSL

Victim Code. In WolfSSL's RSA algorithm, BEEA is used to compute e^{-1} mod λ, where e is the public key, typically a small integer such as 65537. Besides, we have $\lambda = (p-1)(q-1)/G$, where $G = \gcd(p-1, q-1)$.

Method. We synchronize with the victim using the page fault sequence shown in Figure 13(c), and then use a method similar to the one mentioned in Section 6.3 to recover the branch direction during the sub-step process. After obtaining λ, we test all possible G from 2 to λ. For a given G, we let $\Phi(N) = \lambda G$. We let $N = pq$, which is also a part of the public key, and computer the candidate p and q as:

$$p,\ q = \frac{(N+1-\Phi(N)) \pm \sqrt{|N+1-\Phi(N)|^2 - 4N}}{2}.$$

We use the Newton-Raphson algorithm [25] to test whether the recovered values are integers. If they are, we return p and q. Otherwise, we continue testing the next possible G.

Evaluation. We use MDPeek to recover the randomly generated large primes p and q during the RSA key generation process for a 2048-bit key. Similar to MbedTLS attack, we perform 1000 attacks with different keys and use various loads to reduce noise. Each attack takes approximately 880 ms. The success rate for a single load exceeds 95%. When more than 5 loads are used, the attack success rate approaches 100%.

7 Impact and Limitations of MDPeek

7.1 Practical Impact

MDPeek demonstrates a novel side-channel attack against SGX. In addition to bypassing the defense strategies outlined in Table 1, it also has practical implications for certain side-channel defenses and the design of future TEEs.

Impact on constant-time implementation. Some existing defense techniques employ constant-time implementations to mitigate side-channel attacks. For example, the state-of-the-art defense technique Obelix [60] obscures the program's

control flow by reordering instruction sequences. To defend against program counter (PC) leaks, such as the Frontal Attack [44], Obelix places memory access instructions into 64-byte aligned instruction slots, ensuring memory access alignment. However, MDPeek shows that the alignment granularity still needs to be increased. Specifically, the alignment should be raised to 256 bytes.

Impact on future TEEs. MDPeek requires two critical factors. First, the MDU is not flushed during context switches. Second, SGX can be interrupted by the attacker. While SGX-Step, the tool used in this work, is specific to Intel SGX, a recent study [61] shows that similar attack frameworks can also be applied to Intel TDX, the latest version of Intel's TEE. Furthermore, we observe that the MDU design has not been updated in the latest 13th and 14th-generation Intel Core CPUs. Given these factors, Intel's current and future TEEs should still account for the potential threat of MDPeek attacks. We leave the feasibility of MDPeek on Intel TDX and future TEEs for future work.

7.2 Comparisons and Limitations

Noise resistance. Some SGX side-channel attacks require repeated experiments to reduce noise and improve success rates. For instance, the Frontal Attack [44] requires 1000 repetitions to achieve a 60% success rate. This requires that the victim use the same key across multiple executions, such as in RSA decryption, which limits other attack scenarios that rely on single-trace analysis, like RSA key generation.

In contrast, MDPeek demonstrates better noise resistance, achieving a high success rate with a single execution. This is because the state of the MDU is less susceptible to interference from interrupt handling and context switching. To further verify this, we analyze the update of all 256 MDU counters during a page fault and context switch, observing the stability of their counts. Specifically, we find that 210 counters maintain a stable count with a probability of 95%, either remaining unchanged or consistently decreasing by a fixed value. As a result, MDPeek shows superior noise resistance compared to previous research.

Practicality. Compared to other side-channel attacks [41, 44, 51, 57] and transient execution attacks [8, 50, 55] targeting SGX, MDPeek does not require the use of the APIC for single-step execution, nor does it need complex configurations to trigger specific microarchitectural behaviors. This simplifies the attack and enhances its feasibility.

Limitations. Compared to other attacks, MDPeek imposes higher requirements on the victim's code. For example, BPU attacks [14, 65] only require a branch instruction, with no specific requirements for the instructions along the branch path. Similarly, some attacks leveraging cache and prefetchers [9, 40] only require the presence of a load instruction along the branch path. However, MDPeek requires a store-load pair along the branch path, which limits its attack surface and increases the difficulty of software analysis. We leave the development of more efficient and accurate software analysis techniques for identifying MDPeek-usable code to future work.

7.3 Other Security Implications

Other attacks on Intel CPUs. In addition to side-channel attacks targeting SGX, the reverse engineering of the MDU reveals that it can also be updated during transient execution. Therefore, similar to previous studies [5, 47], the MDU can serve as a new covert channel, allowing attackers to leak data during transient execution. Furthermore, we discover that the MDU is time-multiplexed between two hyperthreads on the same physical core. As a result, the MDU could also become a new source of leakage, potentially enabling covert channels or side-channel attacks across logical cores [52].

Similar attacks on other architectures. Recent studies [33, 34] have shown that, similar to the MDU, both AMD and Arm CPUs have implemented components for memory disambiguation. However, unlike Intel, Arm CPUs do not have counters for the MDU [33], while counters on AMD's memory access predictors are selected by physical addresses [34], which may increase the complexity of the attack. We leave the exploration of similar attacks on AMD for future work.

8 Defenses against MDPeek

8.1 Proposed Defense Mechanisms

Under the threat model proposed in this paper, the attacker can disable hardware features such as ASLR [13] and Speculative Store Bypassing Disable (SSBD) [24], making current hardware-based defenses ineffective against MDPeek. In this section, based on our reverse engineering, we present several software-based defenses and evaluate their performance. We implement these defenses in the LLVM backend.

Serialization. According to the MDU's enable logic, both the store and load must be present in the ROB simultaneously for the MDU to trigger an update. Therefore, inserting an `lfence` between the store and load prevents the MDU from updating. We implement this defense in LLVM by inserting an `lfence` either after the store or before the load.

Balancing. Based on the MDU's selection mechanism, MDPeek cannot distinguish between loads with the identical lowest 8 bits of their PCs (i.e., loads aligned to 256 bytes). We place these loads at the entry of basic blocks and use LLVM's API to ensure the basic blocks are aligned to 256 bytes.

Store-to-load coupling. According to the MDU's enable logic, the MDU only predicts and updates when the CPU is uncertain about the data dependence between the store and load. To address this, we insert arithmetic instructions between the store and load to couple them. As shown in Figure 5, we add an `and` instruction, using the store address as the source and the load address as the destination operand, allowing the CPU to recognize the dependence between the store and load. Based on our analysis in Section 4.6, for

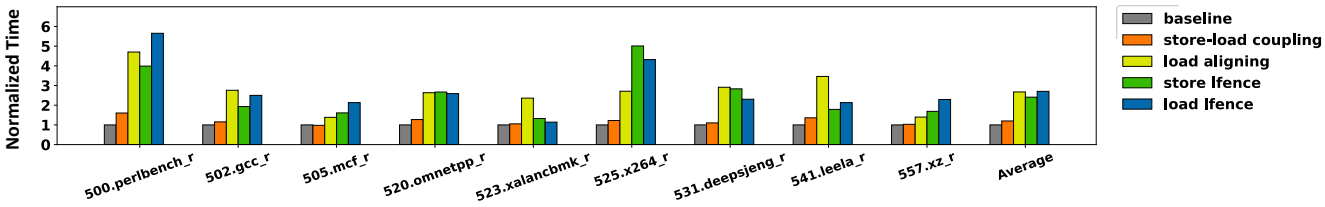

Figure 14. Performance evaluation of defenses. We normalize the execution time of each benchmark according to the baseline (i.e., complied without defenses). The performance overhead of store-to-load coupling is much smaller than other defenses.

each exploitable load, we only need to identify the nearest store and couple them. This method delays the store address generation (i.e., t_{s2} in Figure 4) but does not block out-of-order execution in subsequent stages, thereby minimizing the performance loss.

Evaluation. We evaluate the performance of the above methods using the SPEC 2017 `intrate` benchmark suite. Benchmarks without any defenses serve as the baseline, and we compare the performance overhead of each defense method. The results are shown in Figure 14. The `lfence` defense, which prevents out-of-order execution after the store or load, incurs the highest overhead, with an average performance loss of 140% for store fencing and 170% for load fencing. The balancing defense disrupts the program's spatial locality, reducing the effectiveness of the cache and prefetcher, leading to a performance overhead of approximately 160%. In contrast, store-to-load coupling preserves as much CPU out-of-order execution and cache performance as possible, resulting in an average performance overhead of 20%, a significant improvement of 7 times compared to other defenses.

8.2 Discussion and Future Work

Software-based Defenses. In addition to using the `xor` instruction, there are other ways to explicitly establish data dependence. For example, YSNB [42] employs the `xor` and `lahf` instructions to enforce data dependence by linking registers stored at branch boundaries or the `EFLAGS` register with sensitive data accessed within the branch, thereby mitigating Spectre-V1 attacks [27]. Additionally, other arithmetic instructions can also be leveraged to establish data dependence and prevent MDU updates. Evaluating the performance overhead introduced by different instructions for explicitly establishing data dependence remains an open research question, which we leave for future work.

Hardware-based Defenses. Under the threat model considered in this work, existing hardware-based defense mechanisms are ineffective. For example, while Intel CPUs provide SSBD to mitigate the Spectre-V4 attack by disabling the MDU and speculative execution of the loads, an attacker with root privileges can simply disable this protection. Nevertheless, future CPU or TEE designs could incorporate mitigations against MDU-based side-channel attacks, aiming to prevent MDPeek while minimizing performance overhead.

For example, store and load instructions with unresolved dependence can be dynamically identified during execution. Compared to software-based approaches, this would allow for more precise detection and blocking of vulnerable loads that update the MDU. A similar approach has been proposed for identifying vulnerable loads in Spectre attacks [53].

Besides, for vulnerable loads, a more fine-grained serialization mechanism can be employed to prevent execution or MDU updates. For example, the CSF architecture [54] implements a lightweight `dfence` instruction, which prevents the execution of loads with unresolved memory disambiguation without blocking other instructions. This mechanism can be extended to restrict loads from updating the MDU, thereby enabling more efficient load hardening.

9 Conclusion

In this paper, we present MDPeek, the first side-channel attack that leverages the Memory Disambiguation Unit (MDU). MDPeek is capable of bypassing existing defenses and compromising control flow in SGX, such as balanced branches. To implement MDPeek, we perform an in-depth reverse engineering on the MDU's enable and update logic and develop a code pattern for automatically identifying exploitable loads in Libjpeg, MbedTLS, and WolfSSL. We successfully implement three end-to-end attacks. Finally, we propose a defense to protect the applications within SGX against MDPeek. This research highlights the need for further considerations in defending against control flow attacks in SGX, particularly regarding the potential impact of new leakage sources.

Acknowledgments

We would like to thank Professor Ashish Venkat for his insightful suggestions, Jiapeng Zhou for pointing out some typographical mistakes, and the anonymous reviewers for their valuable feedback, which greatly helped us to improve this work. We also sincerely appreciate Professor Dapeng Ju for his generous help to manage our experimental setup. This work was partially funded by the National Natural Science Foundation of China (Grant No. U23B2041, U24A6009) and the Singapore Ministry of Education (Grant No. T1251RES2403, T2EP20222-0026). Chang Liu is supported by China Scholarship Council (Student No. 202406210249).

A Artifact Appendix

A.1 Abstract

This artifact consists of two main components: first, a proof-of-concept (PoC) code demonstrating that the Memory Dependence Unit (MDU) is not isolated within or outside of SGX; and second, an end-to-end attack on MbedTLS RSA using MDPeek. The PoC code leverages `ecall` to establish a covert channel inside and outside SGX using the MDU, encoding a bit into an MDU counter on each execution. This process transmits the bit from inside SGX to outside. This experiment not only validates the feasibility of MDPeek but also demonstrates how to initialize and probe the MDU counters, as detailed in Section 6.1.

The MbedTLS attack uses SGX-Step to synchronize the attacker and victim. The attacker manipulates the page tables to interrupt the victim's execution at a 4KiB granularity, and initializes and probes the MDU at the appropriate times, as shown in Figure 3. By observing changes in the MDU counters, the attacker infers the branching direction dependent on secret data in the modular inversion algorithm of MbedTLS, thus leaking the large prime numbers p and q, completing the attack described in Section 6.3.

Both the PoC and MbedTLS attack require SGX and must therefore be executed on Intel CPUs that support SGX. The MbedTLS RSA attack specifically requires SGX-Step as the attack framework to allow the attacker to hijack the victim's control flow at the page granularity. As an example, this section describes how to run the artifact on an Intel i7-6700K CPU with the Ubuntu 18.04 operating system and Linux kernel version 5.4.0.

A.2 Artifact check-list (meta-information)

- **Program:** A proof of concept (PoC) demonstrating that the MDU is not isolated inside and outside Intel SGX, along with an end-to-end MDPeek attack against MbedTLS RSA, as described in Section 6.3.
- **Compilation:** GCC version 7.5.0 and G++ version 7.5.0.
- **Binary:** A precompiled static library of MbedTLS version X (replace with version number).
- **Run-time state:** ASLR disabled, SGX-Step enabled.
- **Run-time environment:** Ubuntu 18.04, Linux kernel version 5.4.0, Python 3.12.8, make, sudo.
- **Hardware:** Intel i7-6700K.
- **Execution:** For the PoC, a covert channel is exploited to simulate the MDPeek attack; for the MbedTLS attack, SGX-Step is used to break a list of pages, and the MDU is probed at specific pages to leak the secret-dependent control flow.
- **Output:** For the PoC, a string inside SGX is leaked to the attacker outside SGX and printed on the command line. For the MbedTLS attack, after several attempts, the private big primes p and q are leaked and printed on the command line, which can be further verified through the normal output key file of MbedTLS.
- **Disk space required (approximately):** 3 GiB.
- **Time required to prepare the workflow (approximately):** 2 hours.
- **Time required to complete experiments (approximately):** 1 minute.
- **Publicly available?:** Yes.
- **Code license (if publicly available):** Apache-2.0 License.
- **Archived (provide DOI)?:** 10.5281/zenodo.14776190.

A.3 Description

A.3.1 How to access. The code can be accessed from Zenodo: https://doi.org/10.5281/zenodo.14776190, or from GitHub: https://github.com/CPU-THU/MDPeek.

A.3.2 Hardware dependencies. SGX must be enabled, and specific configurations of SGX-Step are required. To enable SGX-Step, we recommend using devices where SGX-Step has been demonstrated to work. For example, we use the Intel i7-6700K to run the artifact. For more details, please refer to the SGX-Step repository: https://github.com/jovanbulck/sgx-step.

A.3.3 Software dependencies. C and C++ compilers are required to build the attack program. For instance, we use `gcc 7.5.0`, `g++ 7.5.0`, and `make 4.1` to compile the attack program. The SGX SDK is required to build the PoC, while the SGX-Step driver and source code are necessary to build the end-to-end attack program.

A.3.4 Code Organization. The code is located in `/home/mdpeekae/MDPeek`, and it is organized in two main directories, each corresponding to a different experiment.

- **PoC (Proof of Concept) demo of MDPeek**: located in the directory `attack-demo`, containing the PoC attack demonstration to verify the MDU-based side-channel attack (i.e., MDPeek) proposed in our paper.
- **End-to-end attacks through MDPeek**: located in the directory `attack-mbedtls`, corresponding to the MbedTLS attack in the paper in Section 6.3 and containing the setup and scripts of the MbedTLS Attack in Section 6.3.

A.4 Installation

A.4.1 PoC. Please ensure the SGX SDK and SGX driver have been installed before building the PoC. For more details on SGX installation, please refer to https://github.com/intel/linux-sgx. If SGX SDK in installed, update the SGX SDK path in Makefile if necessary. SGX SDK path locates in line 34 of Makefile:

```
SGX_SDK ?= <path-to-sgx-sdk> # SDK path
```

Then we can build the attack framework:

```
make
```

We need to ensure that the least significant 8 bits of the load PC should be aligned, so that the code in the enclave

selects the same MDU counter. Please follow the `readme.md` file to check the alignment. If the loads in the enclave are not aligned with the loads outside, please follow the instructions in the `readme.md` file to adjust them.

A.4.2 MbedTLS Attack. This attack uses the SGX-Step framework to trigger page faults and hijack the control flow of the victim. Please install SGX-Step following instructions in their repository. By default, the SGX SDK will be installed to `/opt/intel/sgxsdk`. After building the SGX-Step, insert the kernel module:

```
sudo insmod <path-to-sgx-step>/kernel/sgx-step.ko
```

Next, update the SGX-Step and SGX SDK paths in Makefile if necessary. SGX SDK path locates in line 34 of Makefile, and SGX-Step path locates in line 77 of Makefile.

To avoid the complicated control flow hijacking setup, update the `libsgx_mbedcrypto.a` in SGX SDK. The default path is `/opt/intel/sgxsdk/lib64`:

```
sudo cp lib/libsgx_mbedcrypto.a
<path-to-sgx-sdk>/lib64
```

Finally, build the attack framework:

```
make
```

A.5 Experiment workflow

A.5.1 PoC. The attacker, located outside SGX, first initializes two MDU counters corresponding to two store-load pairs on the two paths of a secret-dependent branch within SGX. Then, the attacker triggers the execution of the branch via `ecall`. During execution, the branch path is determined by a specific bit from a byte of the secret string. After the branch execution, the attacker regains control and probes the updates to the two MDU counters. The secret bit can be inferred from the update of one of the MDU counters. Each bit of the secret string is leaked through a single trace, meaning each bit is leaked only once.

A.5.2 MbedTLS Attack. Using the method proposed in Section 5, the attacker first analyzes the modular inversion function of MbedTLS to identify exploitable store-load pairs in the paths of the secret-dependent branch. Since the generation of load instructions is closely tied to the compiler version and compilation options, the artifact uses a precompiled static library of MbedTLS as the victim program. The attacker manages the page tables and hijacks the victim's execution through page faults, enabling the MDU side-channel attack. The attacker identifies which MDU counters are associated with the loads in the victim's address space and places a load in their own address space, aligned to 256 bytes, thereby creating a shared MDU counter.

To determine when to initialize and probe the MDU counters, the attacker constructs a fixed page fault chain independent of the secret. Specifically, the attacker triggers page faults sequentially across a set of pages, performing the MDU side-channel attack on specific page faults while using others to obtain the victim's current control flow information. Therefore, the attacker must have prior knowledge of the victim's control flow at the page granularity and disable ASLR. Given the attacker's root privileges, these actions are entirely feasible.

The workflow of the MbedTLS attack is shown in Figure 3. Before reaching the secret-dependent branch and executing the store-load pair, the attacker uses a set of store-load pairs in the attacker's own address space to execute stores and loads at the same address, setting the two MDU counters to 15. After the victim executes the secret-dependent branch, a load on one of the branch paths updates the MDU counter to 14, while the other load is not executed, leaving the corresponding MDU counter at 15. The attacker then probes the MDU counters again to observe which counter is updated, inferring the branch direction. Finally, the attacker uses a Python script to reconstruct the secret values p and q from the branch directions.

A.6 Evaluation and expected results

A.6.1 PoC. To run the PoC, execute:

```
./app
```

The attack procedure will leak the secret string, which will be displayed on the command line. The secret string can be modified to further demonstrate the feasibility of MDPeek. For more details, please refer to the `readme.md` file.

A.6.2 MbedTLS Attack. First, disable ASLR because the attacker has the root privilege:

```
sudo ./scripts/disable_aslr.sh
```

Second, run the attack:

```
python3 attack.py
```

Note that the python version should be later than 3.7. If the attack procedure gets stuck due to a communication issue, please kill the process and try again.

The attack procedure will try at most 10 times to leak p and q of the RSA private keys. When a p and q larger than 0 is leaked, the procedure will print the values and exit. The output of MbedTLS (the ground truth) is recorded in file `rsa_output.txt`. We can print the real p and q from this private key file to verify that the attack is successful.

References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 673–686, 2019.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *Symposium on Security and Privacy (SP)*, pages 870–887, 2019.

[3] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro Cabrera Sarmiento, and Santiago Sánchez-Solano. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *International Journal of Circuit Theory and Applications*, 45(2):199–213, 2017.

[4] Christopher Batten. ECE 4750 Computer Architecture Intel Skylake. https://www.csl.cornell.edu/courses/ece4750/2016f/handouts/ece4750-section-skylake.pdf, 2015.

[5] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 785–800, 2019.

[6] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium (USENIX Security)*, pages 249–266, 2019.

[8] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *European Symposium on Security and Privacy (EuroS&P)*, pages 142–157, 2019.

[9] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 16–32, 2023.

[10] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves. In *USENIX Security Symposium (USENIX Security)*, pages 4051–4068, 2023.

[11] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Symposium on Security and Privacy (SP)*, pages 45–60, 2009.

[12] Travis Downs. Memory Disambiguation on Skylake. https://github.com/travisdowns/uarch-bench/wiki/Memory-Disambiguation-on-Skylake, 2021.

[13] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Annual International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[14] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[15] FVC2004. Fingerprint Verification Competition 2024. http://bias.csr.unibo.it/fvc2004/default.asp, 2004.

[16] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In *Symposium on Security and Privacy (SP)*, pages 2256–2272, 2023.

[17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the European Workshop on Systems Security (EUROSEC)*, pages 1–6, 2017.

[18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium (USENIX Security)*, pages 955–972, 2018.

[19] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment - International Conference, (DIMVA)*, pages 279–299, 2016.

[20] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution. In *Engineering Secure Software and Systems: International Symposium, (ESSoS)*, pages 44–60, 2018.

[21] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 299–312, 2017.

[22] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 321–347, 2020.

[23] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html.

[24] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2024.

[25] Robert I Jennrich and Stephen M Robinson. A Newton-Raphson algorithm for maximum likelihood factor analysis. *Psychometrika*, 34(1):111–123, 1969.

[26] Donald B Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[27] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[28] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium (USENIX Security)*, pages 557–574, 2017.

[29] libjpeg. Libjpeg Version 9f. https://www.ijg.org/files/jpegsrc.v9f.tar.gz, 2024.

[30] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 63–79, 2019.

[31] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD Prefetch Attacks through Power and Time. In *USENIX Security Symposium (USENIX Security)*, pages 643–660, 2022.

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: reading kernel memory from user space. In *USENIX Security Symposium (USENIX Security)*, pages 973–990, 2018.

[33] Chang Liu, Yongqiang Lyu, Haixia Wang, Pengfei Qiu, Dapeng Ju, Gang Qu, and Dongsheng Wang. Leaky MDU: ARM Memory Disambiguation Unit Uncovered and Vulnerabilities Exposed. In *Design Automation Conference, (DAC)*, pages 1–6, 2023.

[34] Chang Liu, Dongsheng Wang, Yongqiang Lyu, Pengfei Qiu, Yu Jin, Zhuoyuan Lu, Yinqian Zhang, and Gang Qu. Uncovering and Exploiting AMD Speculative Memory Access Predictors for Fun and Profit. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 31–45, 2024.

[35] LLVM. llvm-project. https://github.com/llvm/llvm-project, 2024.
[36] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. BITE: Bitcoin Lightweight Client Privacy using Trusted Execution. In *USENIX Security Symposium (USENIX Security)*, pages 783–800, 2019.
[37] Mbedtls. MbedTLS Version 2.6.1. https://github.com/Mbed-TLS/mbedtls/tree/mbedtls-2.6.1, 2017.
[38] Mbedtls. Mbedtls version 3.6.1. https://github.com/Mbed-TLS/mbedtls/tree/mbedtls-3.6.1, 2024.
[39] misc0110. Pteditor. https://github.com/misc0110/PTEditor, 2024.
[40] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 69–90, 2017.
[41] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *USENIX Security Symposium (USENIX Security)*, pages 469–486, 2020.
[42] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *arXiv preprint arXiv:1805.08506*, 2018.
[43] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database Using SGX. In *Symposium on Security and Privacy (SP)*, pages 264–278, 2018.
[44] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *USENIX Security Symposium (USENIX Security)*, pages 663–680, 2021.
[45] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security Symposium (USENIX Security)*, pages 1451–1468, 2021.
[46] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium (USENIX Security)*, pages 431–446, 2015.
[47] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *2021 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 361–374, 2021.
[48] Ludger Rüschendorf. The Wasserstein distance and approximation theorems. *Probability Theory and Related Fields*, 70(1):117–129, 1985.
[49] Michael Schwarz and Daniel Gruss. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. *IEEE Security & Privacy*, 18(5):18–27, 2020.
[50] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 753–768, 2019.
[51] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. Microscope: enabling microarchitectural replay attacks. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 318–331, 2019.
[52] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. SecSMT: Securing SMT Processors against Contention-Based Covert Channels. In *USENIX Security Symposium (USENIX Security)*, pages 3165–3182, 2022.
[53] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 395–410, 2019.
[54] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Mitigating Speculative Execution Attacks via Context-Sensitive Fencing. *IEEE Design & Test*, 39(4):49–57, 2022.
[55] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium (USENIX Security)*, pages 991–1008, 2018.
[56] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX)*, pages 1–6, 2017.
[57] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 178–195, 2018.
[58] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium (USENIX Security)*, pages 1041–1056, 2017.
[59] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 2421–2434, 2017.
[60] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. Obelix: Mitigating Side-Channels through Dynamic Obfuscation. In *Symposium on Security and Privacy (SP)*, pages 189–189, 2024.
[61] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 79–93, 2024.
[62] wolfSSL. wolfSSL Version 5.7.2-stable. https://github.com/wolfSSL/wolfssl, 2024.
[63] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Symposium on Security and Privacy (SP)*, pages 640–656, 2015.
[64] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, pages 719–732, 2014.
[65] Hosein Yavarzadeh, Archit Agarwal, Max Christman, Christina Garman, Daniel Genkin, Andrew Kwong, Daniel Moghimi, Deian Stefan, Kazem Taram, and Dean Tullsen. Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 770–784, 2024.
[66] Jiyong Yu, Trent Jaeger, and Christopher W. Fletcher. All Your PC Are Belong to Us: Exploiting Non-control-Transfer Instruction BTB Updates for Dynamic PC Extraction. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2023.