Adversarial Assertions

Prabhat Mishra
University of Florida, Gainesville, Florida, USA

***Abstract*—Assertions are statements that can be embedded inside any software program or hardware implementation for checking expected behaviors. They are widely used for validation of functional behaviors as well as faster bug localization. There are no prior efforts to investigate whether the assertions can be exploited by an adversary. In this paper, we demonstrate a timing side-channel attack by exploiting the assertion behaviors. Specifically, we show that an adversary can get unauthorized data by exploiting branch prediction methodologies as well as exception handling delay in assertions. Experimental results demonstrate that our attack is successful on multiple Intel and AMD processors. State-of-the-art Spectre fuzzing tools are unable to detect the attack from adversarial assertions. Therefore, the designers should consider the security implications before introducing assertions in their designs. We also propose a mitigation technique to defend against adversarial assertions.**

***Index Terms*—Assertion-based validation, side-channel vulnerability, transient execution attack, hardware security**

I. Introduction

Assertion-based validation [1] is widely used for verifying the implementation. Assertions are primarily used to verify that the program operates as intended by checking for true conditions at runtime. Assertions can also be extended for formal verification to provide mathematical guarantees that a condition will hold true. When these conditions fail, the assertion triggers an exception or halts the program, thus providing a clear signal of unexpected or erroneous behavior. This mechanism is particularly useful during development and testing phases for identifying and debugging errors. However, assertions can also play a significant role in production environments. They enhance system observability and are crucial for diagnosing issues during the operational phase. Specific assertions are retained in the production environment for faster debugging during post-deployment failure analysis. Therefore, it is important to ensure that the assertions do not reveal sensitive data. If an assertion includes sensitive data and this information is leaked through an error message or a log, it could be considered an information leakage. Moreover, behavior of an assertion can be exploited for side-channel attacks.

Side-channel leakage refers to the disclosure of information through indirect channels instead of direct data breaches. These indirect channels may utilize physical side channels (e.g., power, electromagnetic emanation, etc.) as well as micro-architectural side channels. Unlike conventional attacks that target data directly, micro-architectural side channels exploit the inherent behaviors of a processor's architectural components, which are used for out-of-order or speculative execution. These actions frequently result in instructions being executed out of sequence, earlier than anticipated in-order execution, with the aim of enhancing performance. Transient execution occurs when instructions are executed out of order due to an error or unexpected behaviour, leading to inadvertent changes in the microarchitectural states of a CPU. While the outcome of transient execution may not be apparent at the architectural level, it has the capacity to alter microarchitectural states, including cache, load/store buffers, etc. By analyzing the timing and resource utilization differences caused by a transient execution, adversaries can recognize sensitive information using CPU side-channel attacks such as Spectre [2], [3], Meltdown [4], Foreshadow [5], [6] and microarchitecture data sampling (MDS) [7], [8], [9], [10], [11].

In this paper, we utilize assertions to generate a transient execution scenario which leads to a side-channel attack. This attack does not directly breach data but infers it through the observation of system behavior such as computation timing. In modern computing environments, where micro-architectural features like speculative execution are used to enhance performance, assertions could unintentionally facilitate transient execution scenarios. These scenarios occur when instructions are executed out of their intended sequence, potentially altering the state of CPU components like caches or buffers and providing avenues for adversaries to gain access to sensitive data.

Figure 1 illustrates how assertions can be exploited by an adversary to orchestrate a timing attack. Assertion statements inherently behave like conditional branches, which processors must evaluate to determine the subsequent flow of execution. When encountering a branch, the processor uses branch prediction to anticipate the likely path and speculatively executes the subsequent instructions. This speculative execution can lead to transient execution scenarios, where operations are performed on the basis of predictions rather than confirmed outcomes. Such actions can expose sensitive data if the speculative path affects the state of microarchitectural components, such as cache lines. Moreover, when an assertion fails, it triggers an exception, which is a significant deviation from the normal execution flow, typically handled by specialized exception handling routines. This exception handling often incurs a latency penalty, prompting the processor to continue speculatively executing other instructions rather than waiting for the resolution of the exception. This behavior further ex-

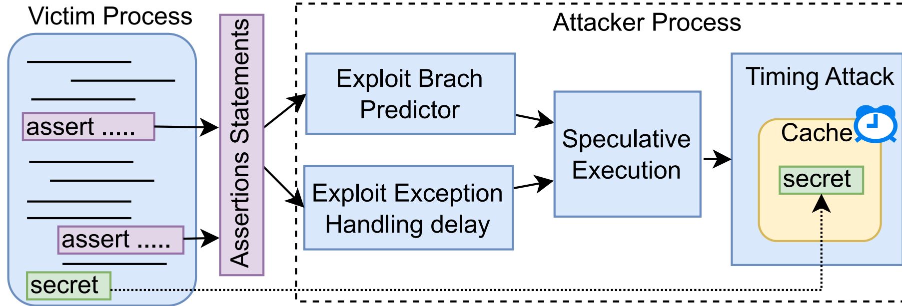

Fig. 1: Overview of our proposed attack using adversarial assertions. A cache timing attack can exploit branch prediction mechanism as well as exception handling delay in assertions to leak unauthorized data.

tends the window for transient executions, potentially altering the state of cache lines that store sensitive data. These transient modifications to the cache state can be exploited through cache timing attacks, such as the Flush and Reload technique [12]. In such attacks, an adversary measures the time it takes to access memory locations; vast differences in these timings can reveal whether data was accessed from the cache.

In this paper, we present adversarial assertions that can be used for side-channel attack to leak unauthorized data. Specifically, this paper makes the following contributions.

- We have identified a new vulnerability in assertion-based validation, referred as *adversarial assertions*.
- We present side-channel attacks by exploiting branch predictor and exception handling delay in assertions. Our proof-of-concept code is publicly available [13].
- Experimental evaluation demonstrates that our proposed attack is successful on multiple machines with AMD and Intel processors.
- We show that state-of-the-art Spectre fuzzing tools cannot detect our attack using adversarial assertions.
- We explore potential countermeasures to defend against adversarial assertions.

This paper is organized as follows. Section II provides relevant background and surveys related efforts. Section III defines the threat model for the attack. Section IV describes the attack using adversarial assertions. Section V explores potential countermeasures. Section VI presents experimental results. Finally, Section VII concludes the paper.

II. Background and Related Work

This section first provides relevant background on out-of-order execution. Next, it surveys related efforts in transient execution attacks and cache timing attacks. Finally, it motivates the need for this work.

A. Background: Out-of-Order Execution

Out-of-order execution is a technique that allows a processor to optimize throughput by rearranging the sequence of instructions, provided the final results remain accurate. The foundation of out-of-order execution lies in the Tomasulo algorithm [14], [15], which utilizes a unified scheduler known as the Reorder Buffer (ROB) to manage the original program's instruction sequence while preserving architectural registers. This process consists of three primary stages. First, an *in-order register allocation and renaming* phase mitigates Write-after-Write (WAW) and Write-after-Read (WAR) hazards through register renaming and dispatches micro-operations (μ-operations) to the ROB. Next, an *out-of-order instruction execution* phase addresses Read-after-Write (RAW) hazards by stalling μ-operations until all necessary operands are available, and then dispatches instructions to respective execution units. Finally, an *in-order retirement* phase releases the instructions in their original order, and writes the results to the respective architectural registers. If transient execution occurs during the retirement phase, the ROB detects it and halts the process, resulting in the flushing of all instructions executed after the transient event, followed by re-execution without speculative actions.

B. Related Work

In this section, we survey related efforts in two broad areas: transient execution attacks and cache timing attacks.

Transient Execution Attacks: Modern processors exhibit security vulnerabilities in their microarchitectural features, leading to the leakage of sensitive information through transient execution. Various attacks, such as Spectre [2], Meltdown [4], Downfall [16] and microarchitecture data sampling (MDS) [7], [8], [9], take advantage of these vulnerabilities. Meltdown [4], is a security vulnerability that takes advantage of out-of-order execution and side-channel attacks to access the memory of the kernel space through a process operating in user space. In contrast, Spectre [2], capitalizes on speculative branch prediction to reach arbitrary memory within a victim process Downfall [16] utilizes the gather instruction in high-performance x86 CPUs to disclose data across user-kernel boundaries, processes, virtual machines, and trusted execution environments. MDS is a Meltdown-type attack that leaks data through internal buffers like store [7] line-fill [8], and load ports buffers [8]. A summary of existing work on transient attacks across different architectures (Intel and AMD) is provided in Table I. It is important to note that the published results suggest the susceptibility of both AMD and

TABLE I: Different type of transient execution attacks on two architectures (AMD and Intel).

	Spectre	Meltdown	Downfall	Store Buffer-based	Fill Buffer-based	Load Buffer-based
Intel	[2]	[4]	[16]	[7], [17]	[8], [9]	[8], [18]
AMD	[2]			[17]		[18]

Intel CPUs to Spectre attacks, whereas Meltdown and MDS attacks are vulnerabilities that exclusively affect Intel CPUs.

Although both Intel and AMD are architectures that belong to the x86 family of instruction set architectures, their implementations differ significantly [19]. Due to this reason, Intel CPUs are more susceptible to MDS/Meltdown-type attacks than AMD CPUs. The major distinction lies in Intel CPUs not flushing underprivileged TLB hits, a characteristic absent in AMD's load execution units/TLB design, where privilege checking for loads is applied differently. This design choice makes AMD architecture show more resistance to Meltdown attack comapred to Intel. MDS attempts to leak data through microarchitectural structures, and while Intel CPUs use various approaches like TSX, simultaneous multithreading, and microcode assists faults for MDS attacks, such errors are handled differently by AMD CPUs, rendering MDS attacks more challenging on AMD machines. Techniques employed by AMD to prevent MDS include TLB flushes across kernel and userspace. Prior efforts have attempted Meltdown/MDS-type attacks on AMD CPUs [17], [18]. Even without the secret residing cache, there are instances where secret can be leaked through buffers. For example, [17] shows leaking of data through store buffer while [18] shows leaking through the load queue.

Cache Timing Attacks: Processors use memory buffers, known as caches, to minimize memory latency, utilizing the spatial and temporal patterns of accessed data. These caches consist of multiple levels, including L1 (primary cache) and L2/L3 (secondary cache), each possessing distinct speed and size attributes. CPU cache timing attacks have emerged due to the significant difference in access cycles between the cache and main memory [20]. These attacks, such as Flush+Reload and Prime+Probe, exploit timing information to distinguish whether data comes from the cache or main memory. In the Flush+Reload method [12], the attacker initiates a cache flush to remove existing data, then reloads the data, and finally measures the cache access time for a cache hit, enabling the identification of data accessed by the victim. In Prime+Probe [21], the attacker fills all cache ways without flushing the cache, subsequently computing the access time to recognize a cache miss, thereby revealing data accessed by the victim. There are variations introduced to Flush+Reload such as Evict+Reload [22] and Flush+Flush [23]. Evict+Reload technique is a modification of the flush process where a specific memory is removed from the cache by accessing addresses mapped to the same cache set. However, a drawback of this method is the prerequisite knowledge of virtual-to-physical address mapping. If the victim has loaded the memory line into the cache, flush takes longer to complete. By substituting Reload with Flush in Flush+Flush, reduces the number of cache misses incurred by reload and avoids triggering prefetching, thereby enabling the attack to evade certain detection mechanisms. In this paper, we employ the Flush+Reload method combined with the speculative execution of an adversarial assertion attack to identify the leak.

To the best of our knowledge, *there are no prior efforts to exploit assertions for leaking sensitive data through cache timing attacks*. This paper is the first attempt at timing side-channel attack by exploiting branch prediction and exception handling delay in assertions.

C. Motivation: Spectre Fuzzing Fails to Detect Adversarial Assertions Attacks

SpecFuzz [24] uses a fuzzing methodology for the dynamic detection of Spectre attacks. This approach introduces speculative execution to the fuzzing process by integrating speculative execution logic into the program during compilation. Subsequently, it relies on the random mutation of program inputs to identify speculative execution errors as they manifest during program execution.

Since the branch prediction exploitation is similar to Spectre attack, we applied the SpecFuzz to detect our attack. When the SpecFuzz is applied to the adversarial assertions attack code, it was not able to detect the branch predictor exploitation behavior. SpecFuzz creates mispredictions by compelling the application to make incorrect branch decisions at each conditional jump. For our attack, this creates an exception with abnormal termination. Unfortunately, SpecFuzz did not handle this termination correctly, resulting in false-negative results even when the adversarial attack was present. While SpecFuzz is effective for identifying Spectre attacks in certain scenarios, it was not able to detect our attack of adversarial assertions.

III. Threat Model

Our threat model assumes that an attacker aims to exploit vulnerabilities in a target system's microarchitecture and speculative execution mechanisms. The goal of the attacker is to gain unauthorized access to sensitive data, by exploiting the speculative execution behavior of the processor. The threat model assumes that attackers possess knowledge of adversarial assertions in the target system and can use this understanding to craft and execute attacks. The attack can occur within the same user space when an attacker exploits assertions in one process to gain access to sensitive information from another process running under the same user account. The attack can also cross user space boundaries, where an attacker would like

```
1 if (x <= buffer_size)
2    int y = buffer[x];
```

(a) Simple if statement

```
1 assert(x <= buffer_size);
2 int y = buffer[x];
```

(b) If statement in Listing 2a converted to an assertion

```
1 jg   .L2
2 movl     -24(%rbp), %eax
3 cltq
4 movl     -16(%rbp,%rax,4), %eax
5 movl     %eax, -20(%rbp)
```

(c) Assembly conversion of Listing 2a

```
1 jle .L2
2 leaq     __PRETTY_FUNCTION__.2331(%rip), %rcx
3 movl     $8, %edx
4 leaq     .LC0(%rip), %rsi
5 leaq     .LC1(%rip), %rdi
6 call     __assert_fail@PLT
```

(d) Assembly conversion of Listing 2b

Fig. 2: An if statement and its equivalent assertion represented using both C and assembly language. (a) and (b) show an if statement and its assertion equivalent, respectively, is C language. (c) and (d) show the assembly versions of the C specifications of (a) and (b).

to leak information from a different user's space or a system process, potentially targeting shared hardware resources.

IV. Adversarial Assertions Attack

In this section, we present our side-channel attack using adversarial assertions. Listing 1 shows a sample code snippet of an adversarial assertion. The second line is an assertion statement which checks if the value of the index x is less than or equal to the size of the $array1$. If x is less than or equal to array1 size, the assertion passes, and the program continues execution. If the assertion condition is not met, where x exceeds the size of array1, and the assertion will fail and raise an exception. The third line retrieves an element from $array2$ using the value of x to index an element in $array1$ and multiplying it by page size.

Listing 1 An example code snippet to illustrate the assertion exploitation for side-channel attack.

```
1 void victim_function(size_t x) {
2  assert(x <= array1_size);
3  val = array2[array1[x] * PG_SIZE];
4 }
```

In Listing 1, the assertion statement can be considered as a speculative gadget where the functionality of the assertion can be exploited to a transient execution behavior, converting the assertion to act as an adversary. The third line of the Listing 1 can be considered as the disclosure gadget which enables access of a victim's confidential information and transmit that information discreetly over a covert communication channel. We first provide insights into the assertion behavior that involves two components: branch prediction and exception handling. Next, we describe how an attacker can exploit branch prediction as well as exception handling of assertions.

A. Assertion Insights

Figure 2 shows the underlying details of an assertion and a simple if statement. Figure 2a shows a conditional statement that checks if the value of the variable x is less than the size of a *buffer*. If this condition is met, y is assigned the value of the element at index x within the buffer. Figure 2c shows the assembly version of the code in Figure 2a. The first line in Figure 2c is a conditional jump instruction. It checks the greater condition and jumps to the label $L2$ if the condition is true. In this case, it checks whether the x value is greater than the $buffer$ size and if the condition is true, jumps to $L2$. Otherwise, the instructions from line 2 - 5 (Figure 2c) are executed, which corresponds to the line 2 in Figure 2a.

Figure 2b illustrates a comparable functionality of the if statement in Figure 2a using an assertion. Figure 2d shows the assembly representation of the assertion statement in Figure 2b. The first line (Figure 2d) is a conditional jump instruction. It checks the less than or equal condition and jumps to the label $L2$ if the condition is true. In this case, it checks whether the x value is less than and equal to buffer size. If the condition is true, it jumps to $L2$ (assertion condition is met). If the assertion fails, it executes line 2 - 6 in Figure 2d. The line 2, 4 and 5 are instructions to gather useful information for debugging, such as function name, line number, etc. The instruction in line 6 calls a function named $__assert_fail$, that is used for handling assertion failures. In other words, this function is invoked when an assertion fails in the program. The specific parameters (arguments) to this function are set up in the previous instructions, which include the condition being tested, the function name, and related information.

This analysis reveals that the functional behavior of an assertion consists of both branch condition and exception handling. The next two sections describe how an attacker can exploit branch condition as well as exception handling in assertions to mount a timing side-channel attack.

B. Attack Utilizing Branch Predictor

This section explores the power of adversarial assertions in exploiting the branch behavior. There can be two types

of conditional branches in a design: direct branch and indirect branch. A direct branch is an instruction that explicitly specifies the jump destination address, either in full or as an offset from a register, within the instruction itself. In contrast, an indirect branch is an instruction that contains a reference to a register or memory address, which, in turn, holds the information about the jump destination address. The assertion statement can be considered as a direct branch and this behavior can be exploited to perform a side-channel attack.

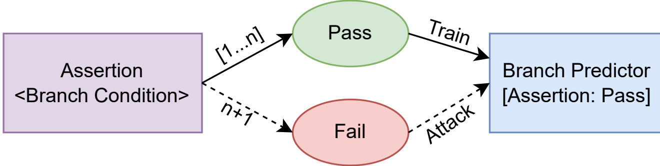

Fig. 3: Use of assertion to train the branch predictor for speculative execution

Figure 3 shows how an assertion can be used to train the branch predictor. First, we repeatedly (1 to n times) execute the assertion such that the condition is correct. The branch predictor learns that this specific branch (the one where the assertion passes) is taken most of the time. After training the branch predictor, we intentionally attempt to access memory beyond the bounds. Since the branch predictor is trained to assume the assertion will pass, we are able to access the out-of-bound memory location. This out-of-bound memory location value will be fetched to the cache, where we can perform a cache timing attack [20].

Listing 2 A sample code snippet that can be used to train the branch predictor [2].

```
1  train_x = rounds % array1_size;
2  for (int i = tries; i >= 0; i--) {
3   x = ((i % 6) - 1) & ~0xFFFF;
4   x = (x | (x >> 16));
5   x = train_x^(x&(malicious_x^train_x));
6   victim_function(x);
7  }
```

Listing 2 shows a procedure to train the branch predictor for several times and then exploit the branch predictor. The attack can be performed for several $rounds$ to get accurate results. The $train_x$ value is derived by getting the modulo of $rounds$ divided by $array1_size$. The value $tries$ decides how many times to train the branch predictor for each round. In line 3 of Listing 2, if i is divisible by 6, the value of x is assigned as $0xFFFF0000$; otherwise, x is set to 0. Line 4 sets x to -1 if i is divisible by 6; otherwise, it sets x to 0. Line 5 sets value of x to $malicious_x$ value if i is divisible by 6; otherwise, x is set to $train_x$. Finally, the *victim_function* is called with the x value.

Table II provides an overview of Listing 2 with example values. When $rounds$ is set to 1 and $array1_size$ is 10, the value of $train_x$ is determined to be 1. If $tries$ is configured as 5, the variable i ranges from 5 down to 0. For each i value that is not divisible by 6 (e.g., i = 5, 4, 3, 2, 1), the value of x is set to 1 (which corresponds to $train_x$). However, when i is 0, x is assigned the value of $malicious_x$ (0xDFF0), which surpasses the boundary of $array1_size$. For values of i from 5 to 1, the assertion is successful, leading to the training of the branch predictor to choose the branch where the assertion passes. However, when i equals 0, the assertion actually fails. Despite this, the branch predictor incorrectly anticipates the assertion to be true, which results in a miss prediction. This miss prediction allows access to $array1[malicious_x]$, an out-of-bounds memory location, causing the data from this location to be loaded into the cache. A cache timing attack [20] can then be used to identify the accessed value. We use Flush+Reload [12] to identify the values in the cache.

TABLE II: Training branch predictor using Listing 2

i	5	4	3	2	1	0
x	1	1	1	1	1	0xDFF0
Assertion	Pass	Pass	Pass	Pass	Pass	Fail

C. Attack Utilizing Exception Handler

This section explores the power of adversarial assertions in exploiting the exception handling time. We are using the exception handling time to speculatively get out-of-bound memory into the cache. In this case, we do not train the branch predictor. We only use the assertion exception as the window. Figure 4 illustrates our proposed attack strategy employing assertion failure. When an assertion failure is triggered, the control flow is directed to an exception handler. Ideally, execution should halt at the time of assertion failure. However, due to the inherent out-of-order execution, instructions after the assertion may still get executed. Importantly, these instructions won't affect architectural states, but they can alter micro-architectural states like the cache. Utilizing this phenomena, we can execute a cache timing attack to find the sensitive data processed post-assertion failure.

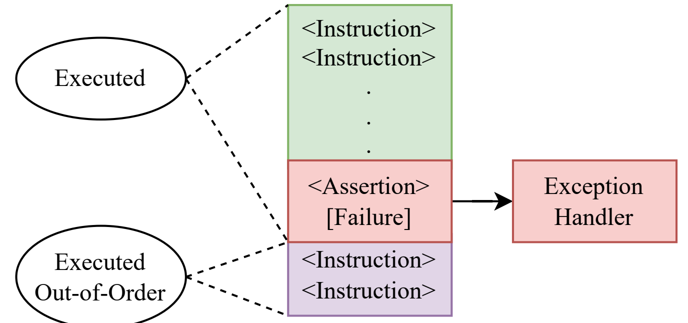

Fig. 4: Assertion failure creates an exception. Modern processors execute instructions (after the assertion statement) out-of-order because the exception handling is delayed. This delay provides an window of opportunity for attackers.

TABLE III: Experimental setup details about the computers and associated microarchitectures

CPU	Year	Microcode	Environment	μ-Arch.	OS
Intel(R) Xeon(R) CPU E5-1620	2012	0x49	Lab	Haswell-EP	Ubuntu 20.04.6 LTS
Intel(R) Core(TM) i7-10510U	2019	0xf4	Lab	Comet Lake	Ubuntu 18.04.6 LTS
AMD Ryzen Threadripper PRO 39995WX	2019	0x830107a	Cloud	Zen+	Arch Linux - rolling
AMD Ryzen 9 5900X	2019	0xa201016	Lab	Zen+	Arch Linux - rolling

We modified the code in Listing 2 to always make assertion failure so that an exception will happen. The for loop in Listing 2 can be updated such that the i value is always divisible by 6. This can be achieved by iterating the for loop such as "*for (int i = tries; i¿=0; i-=6)*". For all the i values this will assign x as $malicious_x$. This will generate an assertion failure which will lead to "abort" exception. Usually exception handling takes more time to process. The processor executes the instructions after the assertion statement out-of-order without waiting for the exception handling result. This temporal window grants an opportunity for an attacker to access the 'array1[x]' value, which resides in an out-of-bound location.

Listing 3 Exception handling for assertion failure

```
1  void unblock_signal(int signum) {
2    sigset_t sigs;
3    sigemptyset(&sigs);
4    sigaddset(&sigs, signum);
5    sigprocmask(SIG_UNBLOCK, &sigs, NULL);
6  }
7  void trycatch_exception_handler() {
8    unblock_signal(SIGABRT);
9    longjmp(trycatch_buf, 1);
10 }
```

When the execution pipeline detects the exception, it would normally result in program termination. However, we prevent program termination by capturing and handling the exception, as demonstrated in Listing 3. The function $unblock_signal$ is responsible for unblocking a specific signal, specified by the $signum$ parameter. In signal handling, processes can block certain signals to prevent them from interrupting or terminating the program's execution. The function $trycatch_exception_handler$ calls the $unblock_signal$ function with $SIGABRT$ as the signal to unblock. $SIGABRT$ is typically used to trigger an abnormal termination of a process and is often used in assertions. By unblocking this signal, the code ensures that it can be delivered and caught in the next step. $longjmp$ function performs a non-local jump. It jumps back to a previously established point in the code.

V. Potential Countermeasures

In this section, we outline potential mitigation techniques for adversarial assertions. Both branch prediction and exception handling delay create speculative behaviour. Therefore, side-channel attacks can be stopped using LFENCE instruction. The LFENCE instruction is a memory fence or a load fence instruction in x86 assembly language. It is used to enforce memory ordering by ensuring that all loads before the LFENCE instruction are fully executed and completed before allowing any subsequent instructions to begin execution. It effectively acts as a barrier that prevents the reordering of memory reads with respect to other instructions, particularly in the context of out-of-order execution. Listing 4 shows the LFENCE mitigation for assertions.

Listing 4 LFENCE mitigation for adversarial assertions.

```
1  void victim_function(size_t x) {
2   assert(x <= array1_size);
3   _mm_lfence();
4   val = array2[array1[x] * PG_SIZE];
5  }
```

VI. Experiments

This section shows the effectiveness of the proposed attack. First, we present our experimental setup. Next, we describe the attack results using both branch predictor and exception handling. Finally, we discuss the mitigation results.

A. Experimental Setup

To demonstrate that adversarial assertions are exploitable in many architectures, we performed our attack on various machines from Intel and AMD, as outlined in Table III. The attack codes, as outlined in Listings 1, 2 and 3, are implemented in C and compiled without any optimization using the GCC compiler. We have made the proof-of-concept code publicly available in [13]. While our attack is successful for all machines in Table III, we have only shown the attack and mitigation results on the Intel Xeon machine.

B. Attack Exploiting Branch Predictor

We have verified that assertion can be used as adversaries. In the attack we conducted, victim has an assertion to check the array size. Attacker train this assertion to pass all the time and then access out-of-bound values as described in Section IV. Finally, the attacker uses Flush+Reload [12] technique to expose the secret value.

Figures 5 shows the average access time (cycles) of the 256 elements after running the attack for 10 $rounds$ and 30 $tries$ per round for 6 bytes. In this scenario, the victim process has a passphrase "Attack", and we are able to identify this phrase

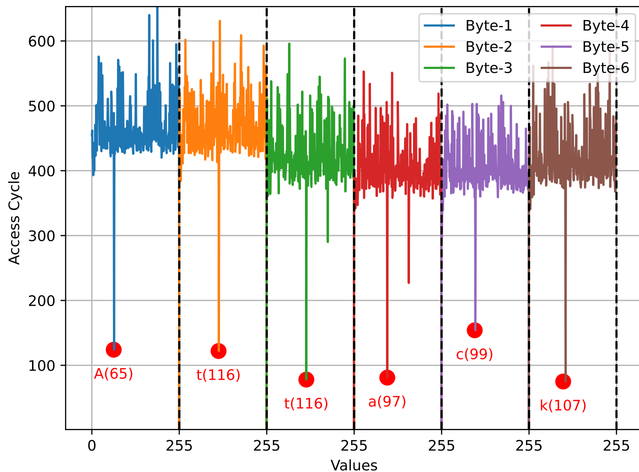

Fig. 5: Access time for the characters in the passphrase ('A', 't', 't', 'a', 'c', 'k') is drastically small compared to the others.

successfully. Figure 5 illustrates that only for characters in the passphrase ('A', 't', 't', 'a', 'c', and 'k'), the access time is drastically small compared to the other values in the buffer. Therefore, it is easy to recognize the values of the bytes. We are able to leak 846 bytes per second with only a 0.1% error rate in byte detection by exploiting the branch prediction.

C. Attack Exploiting Exception Handling

Table IV displays the results of an attack that exploits the delay in exception handling during assertions. In this experiment, the objective is to identify a single byte by executing the attack method described in Section IV. The first column in the table represents the number of rounds used in the experiment. The second column indicates how many of these rounds successfully identified the secret byte out of the total number of rounds. The third column provides the average access time in cycles required to access the value of the secret byte. We are able to leak 12 bytes per second with 6.1% average success rate in byte detection by exploiting the exception handling. In comparison to the results obtained by exploiting the branch prediction behavior of an assertion, this method does not yield consistently persistent results. This is primarily because the delay in exception handling time can be relatively short. Consequently, training the branch predictor has a more substantial impact on the success of the attack.

TABLE IV: Successful attack (wins) per rounds for exploiting exception handling delay for one byte

Rounds	Wins	Average Access Time
10	1	290
100	8	312
1000	69	260
10000	390	272
100000	2171	282

D. Mitigation Results

We examined the inclusion of the LFENCE instruction following the assertion statement as a mitigation against side-channel attacks. Figure 6 shows the average access time for the values without LFENCE and with LFENCE for the byte 1 in Figure 5. The use of LFENCE following the assertion stops the information leakage. Since using LFENCE after the adversarial assertion serializes the instructions and puts an end to speculation.

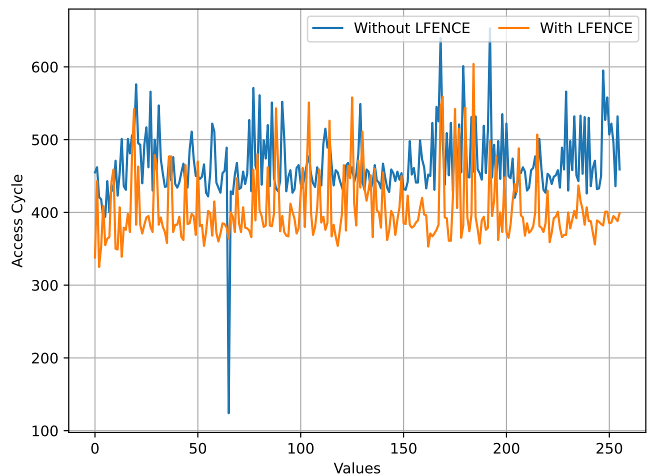

Fig. 6: Average access time with the LFENCE mitigation

To assess the performance impact of using LFENCE, we conducted an experiment and the findings are presented in Figure 7. We computed the average time (cycles) required to execute assertions, both with and without LFENCE following assertions, while systematically increasing the number of assertions using rounds. As illustrated in the figure, adding LFENCE after an assertion leads to an increase in execution time compared to regular execution. According to the experiment, the average performance impact is 3.48 times. It's important to note that the overall performance impact would probably be lower in real-world scenarios, as not all designs involve numerous assertions. Nonetheless, in cases where a design includes a considerable number of assertions, inserting LFENCE between all assertions to safeguard against our attack may not be practical due to the significant performance impact. Therefore, the strategic placement of LFENCE instructions should be performed manually, with a focus on identifying the most vulnerable assertion scenarios.

VII. Conclusion

Assertions are widely used for faster bug localization in software as well as hardware designs. However, the potential for exploitation of assertions by adversaries has not been explored in the literature. We demonstrated a timing side-channel attack on various AMD and Intel machines utilizing assertion behavior via manipulation of branch prediction and exception handling. The state-of-the-art Spectre fuzzing tools

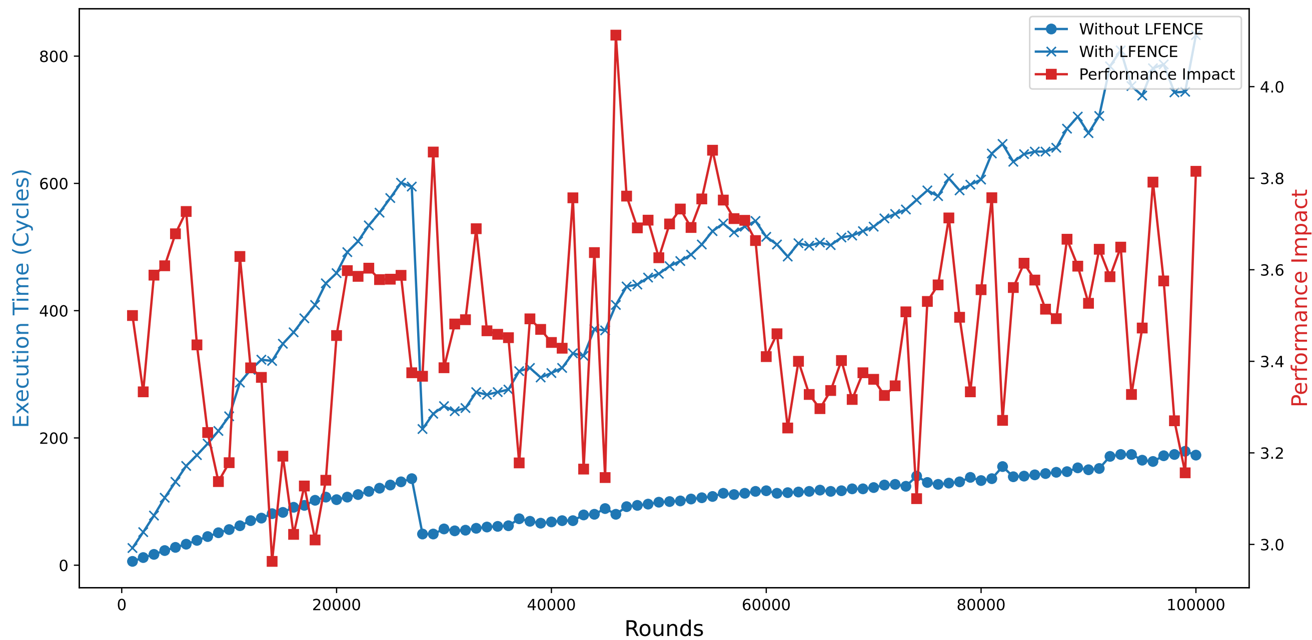

Fig. 7: Performance impact of using LFENCE mitigation

are not able to identify our attack based on adversarial assertions. The developers should be aware of the security implications when incorporating assertions into their designs. Our studies also reveal that lfence-based mitigation can defend against adversarial assertions.

VIII. Disclosure

We submitted proof-of-concept (PoC) exploit for the adversarial assertions to AMD and Intel on October 18, 2023.

References

[1] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. A survey on assertion-based hardware verification. *ACM Computing Surveys (CSUR)*, 54(11s):1–33, 2022.

[2] Paul Kocher et al. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[3] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[4] Moritz Lipp et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[5] Jo Van Bulck et al. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[6] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.

[7] Claudio Canella et al. Fallout: Leaking data on meltdown-resistant cpus. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[8] Stephan van Schaik. RIDL: Rogue in-flight data load. In *Security & Privacy*, 2019.

[9] Michael Schwarz et al. Zombieload: Cross-privilege-boundary data sampling. In *ACM Conference on Computer and Communications Security (CCS)*, pages 753–768, 2019.

[10] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, May 2021.

[11] Jo Van Bulck et al. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[12] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, 2014.

[13] https://anonymous.4open.science/r/Assertion-Exploits-1166/, 2024.

[14] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 1967.

[15] Chao Wang et al. Mp-tomasulo: A dependency-aware automatic parallel execution engine for sequential programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[16] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32th USENIX Security Symposium (USENIX Security 2023)*, 2023.

[17] Saidgani Musaev and Christof Fetzer. Transient execution of non-canonical accesses. *arXiv preprint arXiv:2108.10771*, 2021.

[18] Hasini Witharana and Prabhat Mishra. Speculative load forwarding attack on modern processors. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[19] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, 2, 2012.

[20] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *HASS*, 2018.

[21] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Design Automation Conference*, pages 1–6, 2016.

[22] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive {Last-Level} caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

[23] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: A fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299, 2016.

[24] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetser. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. In *USENIX Security 20*, 2020.