μVM A microarchitectural Intermediate Language for Speculative Execution Testing*

José L. N. C. de Oliveira[1], **Diego L. C. Dutra**[1]

[1]Programa de Engenharia de Sistemas e Computação -
- Universidade Federal do Rio de Janeiro (UFRJ)

{joliveira, ddutra}@cos.ufrj.br

Abstract. *Spectre Branch Target Injection (BTI) is a microarchitectural attack that exploits branch predictors to leak secrets across security domains. While implementation remains tightly coupled to the OS and processor, these vulnerabilities affect x86, ARM, and RISC-V architectures, despite a research bias toward x86 Linux. To cover the largest number of testing scenarios for this type of vulnerability, we present μVM, a microarchitectural intermediate language for speculative execution testing, which we use to test Linux, Windows, x86, and ARM. The framework was also capable of finding 2 bugs in the Linux kernel and undocumented behaviours in Intel CPUs.*

1. Introduction

Since their initial discovery, microarchitectural vulnerabilities have fundamentally altered the landscape of computer architecture, forcing a paradigm shift in how performance-enhancing features are designed for modern microprocessors. At the forefront is the Spectre class of attacks, which exploits the speculative execution of instructions to leak data from memory regions previously assumed to be isolated [Horn , Kocher et al. 2019]. A prominent example is Spectre-BTI (Branch Target Injection), where an attacker poisons the Branch Target Buffer (BTB) — often by manipulating the Branch History Buffer (BHB) — to hijack the control flow of a victim process in a different security context [Barberis et al. 2022]. By injecting malicious entries into these predictors, the attacker forces the processor to speculatively execute a "Spectre gadget": a carefully chosen instruction sequence that encodes sensitive data into a microarchitectural side channel, such as the cache state. This state is subsequently recovered using techniques like FLUSH+RELOAD (F+R) [Yarom and Falkner 2014]. Such exploits enable the exfiltration of secrets across critical boundaries, including hypervisor memory, browser credentials, and kernel-level sensitive data [Lipp et al. 2018, Kocher et al. 2019].

A defining characteristic of these transient execution attacks—including variants such as Retbleed [Wikner and Razavi 2022] and Microarchitectural Data Sampling (MDS) attacks like RIDL [van Schaik et al. 2019], Fallout [Canella et al. 2019], and CrossTalk [Ragab et al. 2021] — is the granular level of microarchitectural knowledge required to identify and validate them [Van Bulck et al. 2020]. While this inherent complexity might appear to be a form of security through obscurity, it represents a trivial

*This project was partially funded by productivity grants from CNPq. (PQ-C: 304308/2025-0), and FAPERJ (JCNE: E-26/204.481/2025)

hurdle for sophisticated adversarial teams. On the contrary, we contend that the significant effort required to port exploits between disparate architectures (e.g., x86, ARM, and RISC-V) is actively detrimental to security. This friction obscures systemic vulnerabilities, increasing the likelihood that critical microarchitectural flaws remain undetected during the design and verification phases of new microprocessor generations.

The complexities of speculative execution mean even widely deployed architectures like x86 lack full remediation against Spectre. We contend that comprehensive mitigation requires transparent implementations and standardized benchmarking across diverse microarchitectures to ensure verifiable protection. To address this gap, we developed μVM, building upon foundational research into speculative execution [Kocher et al. 2019], return-stack vulnerabilities [Koruyeh et al. 2018], and cross-boundary exfiltration [Lipp et al. 2018, Schwarz et al. 2019]. By leveraging these principles, μVM provides a portable testing environment. Through a custom intermediate language and robust runtime, μVM enables architecture-independent exploits, facilitating security validation across varied system configurations. We make the following contributions:

- Creation of a highly accurate framework for Spectre-BTI testing;
- Analyzed a total of 30+ scenarios across 6 different platforms, 2 OSes, and 3 CPU architectures;
- Documented 1 behavior in Intel CPUs related to IBRS;
- Identified 2 bugs related to userspace Spectre-BTI mitigations;
- Compared mitigation differences between Linux and Windows.

The remainder of this paper is organized as follows. Section 2 presents the fundamental concepts and related work. Section 3 details the proposed μVM framework, including its custom intermediate language and runtime environment. Section 4 presents a comprehensive experimental evaluation and analysis of our proposal. Finally, Section 5 provides concluding remarks and outlines directions for future work.

2. Fundamentals and Related Work

This section provides a brief overview of the primary microarchitectural components and their interactions during a Spectre attack. We present a detailed analysis of how an attacker can prime the Branch Prediction Unit (BPU) and exploit the speculative window to exfiltrate sensitive data. Furthermore, we evaluate relevant existing mitigations and conclude this section by discussing prior research on speculative attacks that forms the foundation of our proposal.

2.1. The Branch Prediction unit

Predictions for indirect jumps, such as *jmp rax* in x86 or *bx r0* in ARM, are resolved by the Branch Target Buffer (BTB). The BTB is often a TAGE-like component that stores the targets of indirect branches to enable speculative execution [Luyi Li and Tullsen 2024]. In x86 architectures, the BTB typically utilizes the Branch History Register (BHR) and the current source address for path-based prediction, or solely the source address for IP-based prediction. The distinction among branches sharing the same origin, many processors incorporate a Branch History Buffer (BHB). This hardware component stores a hash of

the most recent execution path to provide context for the predictor. For return instructions, the processor relies on the Return Stack Buffer (RSB) — a dedicated hardware stack that records the return address immediately following a call instruction.

However, the Branch Prediction Unit (BPU) is susceptible not only to branch injection but also to internal state leakage. Because the BPU stores code memory addresses, it can be exploited to bypass ASLR and KASLR [Horn , Evtyushkin et al. 2016, Oliveira and Branco 2023] without requiring a functional Spectre gadget in the victim's code. Furthermore, the BPU can serve as a side channel for more complex attacks, such as Spectre-BTI itself [Mambretti et al. 2019]. While CPU vendors frequently outline this attack model in their security guidance [AMD , ARM a, Intel], it is often treated with lower priority than classic Spectre variants due to its indirect nature.

The fundamental vulnerability in speculative execution arises because these three components (BTB, BHB, and RSB) are shared across multiple execution contexts — including the kernel, user processes, and the hypervisor. If the state of these buffers persists during a context switch, it creates a cross-domain leak. The μVM framework aims to identify not only "classic" Spectre-BTI scenarios but all potential vectors through which these microarchitectural components can leak secrets across security boundaries.

2.2. Available mitigations

Spectre-BTI mitigations rely on writing to Model Specific Registers (MSRs) to enable or disable specific protections, which imposes a heavy performance impact depending on the processor family. Because these MSRs require kernel privileges for modification, the kernel must issue them to (1) protect itself from userspace and (2) isolate user applications from one another. Typically, kernels export these mitigation requests through system calls such as prctl on Linux or SetProcessMitigationPolicy on Windows [Linux , Microsoft]. On x86 architectures, three primary mitigations exist:

1. IPBP protects future branches from previous branches. Linux Kernel issues IBPB when switching to a protected process, for example.
2. STIBP protects a thread from its sibling since processors with SMT share the same BPU.
3. IBRS protects the kernel from userspace. In older versions of IBRS, it was necessary to write 1 into MSR after every kernel switch. For eIBRS/autoIBRS, it's only necessary to write once. If the kernel has support for eIBRS, STIBP is also unnecessary.

On the ARM64 architecture, hardware-level protection is provided through the CSV2 (FEAT_CSV2) framework. While CSV2 serves a purpose similar to Intel's IBRS, it offers a more comprehensive isolation model by strictly restricting the sharing of Branch Target Buffer (BTB) entries based on the current execution context. Specifically, CSV2 utilizes the processor's security state, Exception Level (EL), Virtual Machine Identifier (VMID), and Address Space Identifier (ASID) to partition branch predictor resources [ARM b, Barberis et al. 2022]. By hardware-tagging these entries, ARM64 effectively prevents cross-context pollution, thereby providing intrinsic protection for userspace applications by default. This hardware-enforced boundary significantly reduces the reliance on costly software-based flushes or barriers during frequent privilege transitions.

2.3. x86 Spectre-BTI toy example

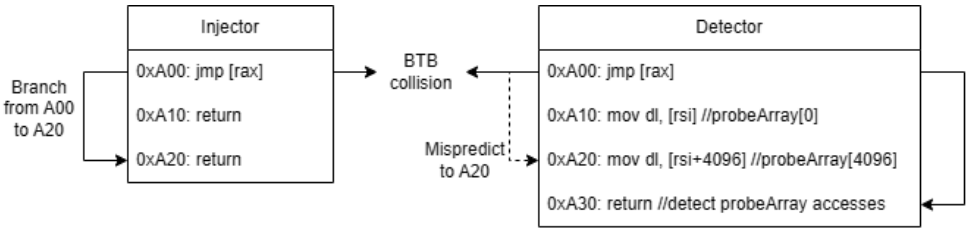


Figure 1. Example for memory mapping Spectre (BTI) detection in x86.

To detect BTB leakage as illustrated in Figure 1, the Injector context first initializes with a secret string and selects a single bit based on the current time. Depending on this bit's value, the Injector loads the rax register with either $0xA10$ or $0xA20$. Simultaneously, the Detector context executes an identical branch at the same virtual address, causing the hardware to mispredict the destination. The processor speculatively fetches and executes a mov dl instruction from the mispredicted address, which accesses probeArray at either index 0 or 4096. Following the misprediction recovery, the Detector performs a Flush+Reload side-channel attack to identify the accessed array index, thereby inferring the secret bit. By looping this process, the Detector reliably recovers the full secret string without requiring a specific Spectre gadget within the victim domain.

2.4. Related Work

The Meltdown researchers [Lipp et al. 2018] demonstrate how out-of-order execution allows attackers to leak kernel memory. The root cause lies in the CPU's delayed enforcement: permission checks for a faulting load only occur at instruction retirement. In the meantime, the processor speculatively executes subsequent (transient) instructions, which leave observable microarchitectural side effects. Because older kernels map kernel memory into the user process address space (marking it non-user-accessible), a userspace attacker can exploit this window. By suppressing the resulting segmentation fault—via an exception handler or forking—the attacker performs a load targeting kernel space and encodes the result into a side channel (such as Flush+Reload). This process creates an attack capable of reading kernel memory at high speeds.

Spectre attacks exploit the Branch Prediction Unit (BPU) to redirect execution flow across domain boundaries. Researchers typically categorize these vulnerabilities into four primary variants: Spectre v1 (Bounds Check Bypass), v2 (Branch Target Injection or BTI), v3 (Rogue Data Cache Load/Meltdown), and v4 (Speculative Store Bypass). To execute a Spectre v1 attack, an adversary trains the BPU to predict that a conditional branch — such as an array bounds check—will always evaluate to true. This training forces the processor to speculatively execute the out-of-bounds access before the hardware enforces the check. In their foundational work, Kocher et al. [Kocher et al. 2019] demonstrate how Spectre v1 bypasses JavaScript array bounds to leak memory from a browser process, while Spectre v2 successfully extracts data across process boundaries and from KVM hypervisors. Furthermore, attackers can manipulate return instruction predictions by targeting the Return Stack Buffer (RSB) [Koruyeh et al. 2018] or the Branch Target Buffer (BTB) via Retbleed techniques [Wikner and Razavi 2022].

The authors of Attacking branch predictors to bypass ASLR [Evtyushkin et al. 2016] exploit the BTB as a side channel to infer ASLR and KASLR by detecting collisions between direct unconditional branches. Their research reveals that these collisions introduce an 8-cycle bubble in the branch instruction's execution time. This technique leaks KASLR in approximately 60 milliseconds, though it offers limited bit coverage for userspace ASLR. Building on these concepts, Ret2ASLR [Oliveira and Branco 2023] incorporates the Spectre-BTI attack in the reverse direction to reliably leak userspace ASLR in under 10 seconds.

Microarchitectural Data Sampling (MDS) attacks represent a distinct class of speculative execution vulnerabilities that target internal CPU structures, such as the Line Fill Buffer (LFB), load ports, and store buffers. While Spectre attacks rely on mistraining branch predictors to execute gadgets within a victim's domain, MDS attacks leverage residual data residing in shared microarchitectural components to influence the attacker's transient execution state. This allows an adversary to sample in-flight data across security boundaries without requiring specific code sequences in the victim process.

Load ports facilitate memory requests by dispatching effective addresses to the cache hierarchy. When an L1d cache miss occurs, the LFB tracks the request until the data returns from higher memory levels. The Rogue In-Flight Data Load (RIDL) attack [van Schaik et al. 2019] demonstrates that transient instructions can access these LFB entries even when they belong to separate security contexts, such as sibling threads. Similarly, store buffers optimize performance by enabling asynchronous writes and store-to-load forwarding. The Fallout [Canella et al. 2019] reveals that when a faulting load fails address translation—often due to supervisor protections or SMAP violations—the forwarding logic incorrectly falls back to matching only the lower address bits. This architectural flaw enables an attacker to speculatively recover recently written data from the kernel or other isolated domains.

3. μVM

To maximize coverage across the widest possible array of attack scenarios, the framework decomposes the testing process into three specialized core components: the runtime environment (RTE), the μVM compiler, and a privileged helper driver. This modular approach enables decoupling the component to be tested (specified by the μVM code) from the scheduling and mitigations specified by the RTE.

The RTE is a high-performance, portable C executable engineered for seamless operation across both Linux and Windows. It serves as the orchestrator for the entire experimental lifecycle—dynamically toggling system-level mitigations, managing memory allocation, and ensuring execution consistency through precise CPU pinning. Crucially, the RTE integrates a robust FLUSH+RELOAD cache side-channel implementation to reliably extract the targeted secret bits with high fidelity.

We achieved efficiency and cross-architecture compatibility via the μVM compiler. By utilizing a custom Intermediate Language (IL), the framework allows us to deploy a single, architecture-agnostic test case across x86 and ARM64 targets. This Python-based compiler generates precise memory mappings of code sections that are instantly

consumed by the RTE, enabling rapid, just-in-time unit testing without the overhead of manual architecture-specific porting.

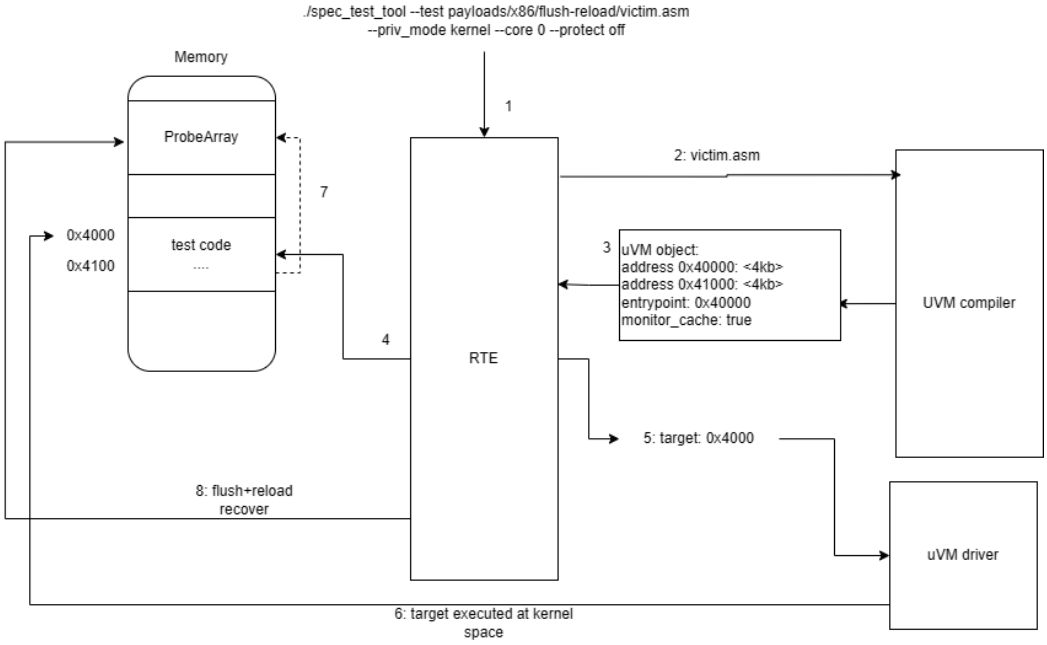


Figure 2. μVM toolkit diagram.

Finally, to bridge the gap between privilege levels, the kernel driver provides a specialized /proc interface designed for deep architectural introspection. By executing user-mode pointers directly in kernel space, the framework bypasses traditional isolation barriers. This methodology requires deactivating KPTI via Linux command-line configurations and disabling SMEP and SMAP (on x86) or PXN and PAN (on ARM64) at runtime. These modifications create a unified testing environment where researchers can analyze kernel-space behavior with the same precision and transparency as user-space code, effectively isolating the processor's privilege level as the experiment's only variable. Figure 2 illustrates the communication among these components, while the following steps detail the communication flow:

1. Arguments are passed to RTE, specifying which test is executed. The selected protections are set, and the task is bound to the specific core. If the mitigation is enabled, the syscall responsible for enabling the mitigation is called;
2. The RTL runs the μVM compiler with the unit test;
3. The μVM generates a memory map containing the test code in the native language and an entry point for the test;
4. The RTL loads the memory map into its current process;
5. If execution mode is kernel, it passes the entry point to the auxiliary driver. If execution mode is user, RTE executes the pointer itself and skips step 8;
6. The driver executes the test code with kernel privilege, but using a userspace virtual address, similar to a ret2usr [Kemerlis et al. 2012] attack;
7. The test code performs a branch and may speculative execute a load to probe an array;

8. After restoring to userspace context, RTE performs a FLUSH+RELOAD attack to detect if the probe array was accessed, allowing to infer the secret injected in the BPU.

3.1. μVM IL language

While these attacks target specific micro-architectural components, the mechanisms governing indirect branch prediction remain remarkably consistent across modern architectures. High-performance implementations in x86, ARM64, and RISC-V typically utilize a TAGE-like structure for the Branch Target Buffer (BTB), a hardware stack for the Return Stack Buffer (RSB), and a dedicated register to store a hash of recent execution history, known as the Branch History Buffer (BHB). The instruction sets across these platforms exhibit a similar functional symmetry; the BTB can be influenced via branch instructions, the RSB is manipulated through 'call' and 'return' pairs, and the BHB state can be shifted by conditional branches—even when the underlying hardware update function remains proprietary and opaque. Table 1 enumerates all the IL instructions implemented.

The primary implementation challenge lies in the reliability of the Flush+Reload (F+R) side channel across different ISAs. On x86 architectures, the process is straightforward, providing high-precision timing via the RDTSC instruction and explicit cache line eviction through CLFLUSH. ARM64 offers similar capabilities, using 'DC CIVAC' as a functional equivalent to CLFLUSH and the CNTVCT_EL0 (Virtual Count Register) as a high-frequency clock. Although this register operates at a lower base frequency compared to x86, it remains sufficiently precise for effective side-channel analysis.

In contrast, ARMv7 presents significant constraints as it lacks a direct user-space instruction for cache eviction. In this environment, the framework must resort to cache thrashing—filling the cache with extraneous data until the target line is evicted based on the hardware's replacement policy. Furthermore, due to the absence of an accessible high-precision cycle counter, the implementation must fall back to the clock_gettime Linux syscall to achieve nanosecond-level resolution. To implement a robust Spectre-BTI proof of concept, as illustrated in Figure 2.3, these platform-specific primitives are abstracted into the following high-level logic:

1. Execute a branch in Injector context;
2. Execute a similar branch in the Detector context;
3. Measure side effects on the cache to infer misprediction.

Since Spectre-BTI attacks are highly sensitive to virtual address mapping, the μVM language provides precise control over the memory placement of instructions. This is managed via labels preceding each instruction; during native compilation, the compiler scales these labels by the cache line size. This ensures that the μVM compiler aligns each Intermediate Language (IL) instruction to occupy exactly one cache line, preventing unintended aliasing or overlap. To minimize noise introduced by Translation Lookaside Buffer (TLB) misses, most tests are constrained within a single memory page (labels 1 - 63). This isolation is critical for kernel-level execution, where a page fault would result in a kernel panic due to the absence of a registered page fault handler within the restricted test context. For communication with the Runtime Environment (RTE), the final 256 bytes ($0x100$) of the memory page are mapped to a dedicated I/O structure. This mapping enables the exchange of the probe array pointer and the retrieval of timing metadata whenever the GET_TIME instruction is invoked.

```
1:  CLEAR_BHB

3:  BRANCH_PTR 11
4:  FINISH
5:  FINISH

11: PTR (4+#SECRET_BIT)
```

(a) Injector Code

```
1:  CLEAR_BHB
2:  FLUSH 10
3:  BRANCH_PTR 10
4:  SIGNAL 0
5:  SIGNAL 1
6:  FINISH
10: PTR 6
```

(b) Detector Code

Figure 3. Spectre-BTI unit test written in uArch language

Using this IL representation, the Toy example, previously presented in Section 2.3, could be written like the unit test in Figure 3.

Table 1. Examples of uArch instructions.

uArch IL instruction	Effect
FLUSH(label)	Invalidates cache line of `label`.
MACCESS(label)	Reads memory at `label` into cache.
SIGNAL(idx)	Sends a cache signal for the Flush+Reload side channel on index `idx`.
SET_BHB(value,branches)	X86 exclusive. Uses the already reversed engineering update function for BHB to store the exact value in the BHB.
CLEAR_BHB	Equivalent to a empty for loop with 300 iterations that sets the value of the BHB to an unkown, but replicable value.
PTR(label)	Declares a pointer to another label
BRANCH_PTR(label)	Indirect branch using a pointer
BRANCH_RET(label)	Indirect branch using a return. Top of the stack is automatically flushed before the branch.
GET_TIME(index)	Store current time in UVM struct in field offseted by index.
PUSH_RSB	Add next address to return stack buffer.

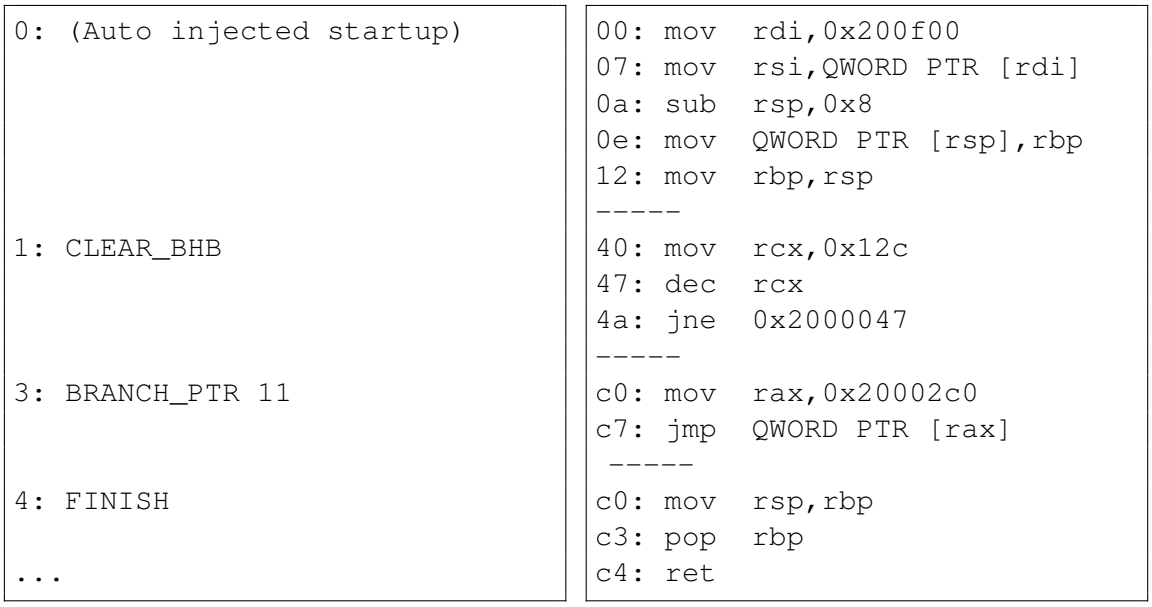
3.2. μVM Runtime

The UVM Runtime is responsible for creating the combinations necessary for multiple tests. Each test is a pair of contexts with the parameters below, per process.

3.2.1. Runtime Mitigations

The mitigations parameter chooses what mitigation is attempted by the userspace application. Enabling the mitigation is expected to make the OS issue IBPB writes and enabling STIBP during context switches between the protected process. The following mitigations are available in the RTE.

1. Off: No mitigation is attempted. The process executes as is, using the default behavior for the target OS.



(a) Original μVM code **(b) x86 compiled μVM code.**

Figure 4. Generation of native code for unit tests.

2. On: In Linux, the target invokes *prctl* with the option *PR_SPEC_FORCE_DISABLE* the first time the process executes. In Windows systems it calls *SetProcessMitigationPolicy*.
3. Switch (Linux exclusive): After enabling the mitigation with *prctl* and the option *PR_SPEC_DISABLE*, Linux allows to remove the mitigation via textitprctl and the option *PR_SPEC_ENABLE*. This test turn on the mitigation before the branch and to turn it off after the branch.
4. Exit: This test is the same as mitigation = On, but the process is recreated multiple times. This was done to check if the scheduler issues the mitigation not only during context switches between process, but also after a process dies.

3.2.2. Runtime privilege mode

This parameter chooses the privilege level at which the payload is executed. This option is used to test for injections &/or leakings that should be mitigated by IBRS/eIBRS/autoIBRS

1. Userspace: The branch is performed inside the context of a process.
2. Kernel: The branch is performed inside the context of kernel, but still using the virtual address of the process.

3.2.3. Runtime scheduling mode

This parameter controls the scheduling of the payloads. Intramode is used as a sanity check for testing that the attack works since most mitigations do not prevent mispredictions, while process is used for evaluating userspace mitigations.

1. Intramode: Runs both injector and detector payloads in the same process. .
2. Process: Runs the injector and detector payloads in separated process scheduled to the same core.
3. Process-SMT: Runs the injector and detector payloads in separated process scheduled to sibling cores on processors with SMT support.

3.2.4. Runtime Recover strategy

This parameter chooses the side channel for the attack.

1. Speculative execution F+R (default): Detects if the speculated instruction was executed via a probe array access. More reliable than the methods below.
2. Speculative fetch: Detects the speculated instruction was fetched to instruction cache but not necessarily executed.
3. Contention: Detects if the current used BTB entry was evicted by another context.

3.2.5. Payload

This parameter is the μVM IL code to be executed.

1. BTB: Detects if one context can speculate to the address injected into the BTB by another context. It uses the injected pointer in the BTB as a side channel to leak the secret.
2. RSB: Detects if one context can speculate to the address injected into the BTB by another context. It uses the injected pointer in the RSB as a side channel to leak the secret.
3. BTB-conten.: Detects if two contexts are competing the same BTB entry, independent of the value being used for prediction. It uses the time difference of the branch instruction as a side channel to leak the secret. This payload uses contention as recover strategy.

4. Results

The experimental evaluation was conducted across all available testbeds. Initially, comprehensive system metadata was collected for each platform, including CPU architecture, kernel version, and the status of enabled Spectre-BTI mitigations. In total, 234 tests were performed across 6 distinct platforms, spanning three architectures (x86_64, ARMv8, and ARMv7) and two operating systems (Windows and Linux).

The testing infrastructure comprised a heterogeneous mix of cloud-based instances and bare-metal processors. This research presents a curated compilation of the most significant results derived from this data collection. To provide a structured analysis, the experiments were categorized into four primary groups based on their security boundaries: intramode, inter-process, process isolation, and kernel isolation. The specific hardware and software configurations for these platforms are detailed in Table 2.

Table 3 presents the results for intramode evaluations, which determine if the proposed attacks are feasible on each target architecture. Since most mitigations do not

restrict execution within a single security domain, intramode execution serves as a sanity check for the experimental setup's integrity. We define an attack as successful when the framework accurately exfiltrates at least 50% of the secret bytes. Among the evaluated architectures, only the ARMv7 platforms consistently failed this baseline check. This systemic failure likely stems from the significant noise and timing variance that occurs without a dedicated architectural cache flush instruction. Consequently, the framework utilized cache eviction strategies to control the cache state, a method that proved less reliable for precise side-channel timing than the flush primitives available in x86_64 and ARMv8.

Table 2. Tested platforms

Platform	Kernel Version	Mitigations
i5-3330 (Baremetal Desktop)	Linux 6.17.5	Retpolines; IBPB: conditional; IBRS_FW; *1 STIBP: disabled; RSB filling; PBRSB-eIBRS: not affected; BHI: not affected;
i7-13620H-wsl (Baremetal desktop, Windows Subsystem for Linux)	Linux 5.15.167.4-WSL2	Enhanced / Automatic IBRS; IBPB: conditional; RSB filling; PBRSB-eIBRS: SW sequence; BHI: BHI_DIS_S;
i7-13620H-win (Windows Baremetal Desktop)	Windows 10.0.26200.8246	BTIHardwarePresent; BTIWindowsSupportPresent; BTIWindowsSupportEnabled; BTIKernelImport OptimizationEnabled;
gc-n1-skylake (Google cloud N1 instance with Intel Xeon CPU)	Linux 6.1.0-44-cloud-amd64	IBRS; IBPB: conditional; STIBP: disabled; RSB filling; PBRSB-eIBRS: not affected; BHI: SW loop (KVM);
SDM450-js8 (Galaxy SM-J810M, bare metal mobile)	Linux 3.18.124	–
A76-rbpi5 (Rasberry Pi 5 bare metal)	Linux 6.17.0-1003-raspi	CSV2 (no BHB);

Note 1: IBRS_FW mitigates only firmware. Kernel space is expected to be vulnerable.

Inter-process evaluations (Table 4) test whether two distinct processes running on the same machine can exfiltrate secrets via speculative side channels. In this scenario, researchers typically encounter the default mitigations that the kernel or hardware architecture implements. For example, our Linux environment utilizes specific Spectre-RSB mitigations; the kernel explicitly implements RSB stuffing within arch/x86/entry/entry_64.S to prevent branch predictions from bleeding across context switches. Conversely, Microsoft provides no official documentation confirming whether Windows employs a similar mechanism, which complicates the verification of its inter-process isolation parity.

Table 5 and Table 6 present Process Isolation and Kernel Isolation evaluations, which verify whether userspace and kernel-level mitigations function as intended. We launch pairs of processes with mitigations enabled, expecting a total absence of data leak-

Table 3. Intra mode checks. Mitigations are default. This checks test if attack is feasible on the architecture.

Platform	Component	Hit rate	Miss rate	Leaked bytes
i5-3330	BTB	77.09%	0.00%	10/10
i5-3330	Cache (F+R)	100.00%	0.00%	10/10
i5-3330	RSB	0.00%	0.00%	-
SDM450-js8	BTB	10.50%	18.25%	- *1
SDM450-js8	Cache (F+R)	75.67%	10.10%	10/10
A76-rbpi5	BTB	29.83%	0.19%	10/10
A76-rbpi5	Cache (F+R)	99.99%	0.00%	10/10
A76-rbpi5	RSB	0.00%	0.00%	- *2
i7-13620H-wsl	BTB	28.38%	0.00%	10/10
i7-13620H-wsl	RSB	90.36%	0.00%	10/10

Note 1: Besides F+R attack being feasible on SDM450-js8, Spectre was unsuccessful.
Note 2: This machine seems unaffected by RSB injection.

Table 4. Inter-process check. Mitigations are default. All tests were run using separate processes, all pinned to core 0.

Platform	Test	Hit rate	Error rate	Leaked bytes
i5-3330	BTB-execution	94.52%	0.00%	10/10
i5-3330	RSB	0.00%	0.00%	-
A76-rbpi5	BTB-execution	0.00%	0.00%	-
i7-13620H-win	BTB-conten.	42.38%	16.91%	10/10
i7-13620H-win	BTB-execution	74.14%	0.00%	10/10
i7-13620H-win	RSB	99.64%	0.00%	10/10 *1
i7-13620H-wsl	BTB-conten.	46.00%	16.06%	10/10
i7-13620H-wsl	BTB-execution	91.10%	0.00%	10/10
i7-13620H-wsl	RSB	0.00%	0.00%	- *1

Note 1: RSB differences discussed in conclusion.

age across all configurations and security boundaries. In general, Spectre-BTI mitigations fail to guarantee complete protection against cross-component data exfiltration; instead, they only ensure that the processor will not execute speculative injections from an unsafe domain into a protected one. The lack of formal documentation regarding these obscure internal mechanisms results in non-standardized behavior across different platforms. Consequently, our test results reveal several unexpected behaviors where side channels remain visible despite the active application of mitigations.

4.1. Mitigation switch bug & Exit bug

The Linux kernel manages Spectre v2 protections through Thread Information Flags (TIFs). Specifically, the ib_prctl_set function toggles the spec_ib_disable flag to enforce

Table 5. Process isolation checks. All tests were run using separated processes, all pinned to core 0. Mitigations vary

Platform	Test	Injector Mitigation	Detector Mitigation	Leak rate
i7-13620H-wsl	BTB-conten.	on	-	10/10 *1
i7-13620H-wsl	BTB	exit	-	10/10 *2
i7-13620H-wsl	BTB	-	on	-
i7-13620H-wsl	BTB	on	-	-
i7-13620H-wsl	BTB	switch	-	10/10 *3

Note 1: Results discussed in BTB contention attack
Note 2: Results discussed in exit vulnerability
Note 3: Results discussed in switch vulnerability

Table 6. Kernel isolation checks IBRS. All tests were run using 2 separated processes, pinned to core 0, except in A76-rbpi5 which the test ran on Intramode.

Platform	Test	Injector	Detector	Chars leaked
gc-n1-skylake	BTB	-	Kernel	-
gc-n1-skylake	BTB	Kernel	Kernel	-
gc-n1-skylake	BTB	Kernel	User	7/10 *1
A76-rbpi5	BTB	-	Kernel	-
A76-rbpi5	BTB	Kernel	Kernel	7/10
A76-rbpi5	BTB	Kernel	-	-

Note 1: Results discussed in IBRS leaking

mitigations during context switches. However, architectural gaps remain; versions prior to 6.1.2 failed to issue an Indirect Branch Predictor Barrier (IBPB) immediately, leaving a brief vulnerability window. While developers addressed this, a critical flaw persists: the kernel does not issue an IBPB upon disabling the mitigation or during process termination (the "Exit Bug"). These omissions leave residual data in the Branch Target Buffer (BTB), which facilitates inter-process leakage as the system exposes secrets from a previous security context to the next scheduled process. Despite the severity of these findings, vendor responses highlight a disconnect between hardware expectations and OS implementation. Furthermore, a fix for the "Exit Bug" remains non-trivial and unproposed.

4.2. IBRS leaking

In certain Intel microarchitectures (specifically Skylake and its predecessors), IBRS appears to provide asymmetric protection: it shields the kernel from userspace injections but fails to protect userspace from kernel-originated injections. This asymmetry potentially exposes kernel-level secrets, such as the KASLR (Kernel Address Space Layout Randomization) offset, through speculative leakage. Notably, employing Retpoline as a standalone mitigation is similarly insufficient, because return instructions also influence the Branch Target Buffer (BTB), kernel-side predictions can still "bleed" into a userspace context [Wikner and Razavi 2022].

4.3. BTB contention attack

In several architectures, it is possible to leverage execution delays in branch instructions to infer data across different security contexts. In affected processors, this behavior suggests that even when mitigations are enabled, logical contexts may still share Branch Target Buffer (BTB) entries, though these entries are likely tagged for specific protected domains (e.g., via a Thread Bit or ASID bit). To exploit this, a sophisticated adversary would need to identify a side channel where an indirect branch's execution depends on a secret, and then determine the specific BTB inputs—such as the source address and Branch History Buffer (BHB) state—required to force a collision. While a practical end-to-end exploit may be improbable, observing this behavior provides critical insight into the underlying hardware implementation of the mitigation.

4.4. RSB leaking on windows

Documentation regarding RSB stuffing on Windows remains notably obscure, as security vendors and the OS developer provide conflicting reports. A Bitdefender whitepaper asserts that up-to-date versions of both Windows and Linux actively implement RSB

stuffing alongside SMEP and SMAP protections [Bitdefender]. Conversely, Microsoft's official stance suggests they have not implemented the measure in userspace; their documentation notes that while certain modern CPUs require software-level RSB stuffing to prevent underflows, they consider a general implementation "nontrivial" for software to perform [Microsoft].

Our experimental evaluations resolve this ambiguity by demonstrating that RSB stuffing is not active by default for userspace applications on Windows. This gap in protection was empirically detected on the i7-13620H-win platform, which our tests showed to be susceptible to RSB-based attacks specifically under the Windows environment. These results corroborate Microsoft's skepticism regarding the complexity of the mitigation while highlighting a persistent vulnerability window in modern Windows-based systems that is effectively addressed in Linux.

5. Conclusion and future works

Despite extensive research and patching, the high-complexity x86 environment has not yet been fully remediated against Spectre-class vulnerabilities. This inherent complexity often results in subtle edge cases that remain improperly addressed. Furthermore, the lack of transparency regarding the internal mechanisms of hardware mitigations exacerbates the problem; current mitigation guidelines often over-abstract the actual implementation and its specific security guarantees. This abstraction frequently leads vendors to overlook vulnerability classes that are not directly related to branch injection. Finally, the persistent misalignment between CPU vendors, kernel maintainers, and user-space developers creates a fragmented landscape, providing a fertile environment for critical errors in mitigation deployment.

By decoupling exploit logic from architectural specifics, μVM overcomes the inherent complexity of transient execution analysis. Our framework leverages an intermediate language and a robust runtime (RTE) to provide a portable, standardized workflow for Spectre-BTI testing. This approach enables the creation of large-scale, multi-scenario evaluations that span diverse computational configurations, transforming the validation of speculative execution mitigations from a manual, architecture-specific hurdle into a transparent and verifiable process.

Future work will expand the framework's depth through three primary avenues. First, developing a dedicated Windows driver will provide kernel-level security validation parity with our Linux implementation. Second, we will utilize FPGA-based prototyping to synthesize advanced RISC-V cores, enabling rigorous speculation testing on hardware currently limited by market availability or cost. Finally, we aim to automate Spectre-class gadget discovery by leveraging our intermediate language to identify cross-architecture vulnerabilities before production.

References

AMD. Amd64 technology indirect branch control extension. `https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/white-papers/111006-architecture-guidelines-update-amd64-technology-indirect-branch-control-extension.pdf`.

ARM. Arm cpu security bulletin: Spectre/meltdown. https://developer.arm.com/documentation/110280/latest/.

ARM. Spectre-bhb: Speculative target reuse attacks. https://community.arm.com/support-forums/f/architectures-and-processors-forum/57489/more-details-on-csv2.

Barberis, E., Frigo, P., Muench, M., Bos, H., and Giuffrida, C. (2022). Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security*. Intel Bounty Reward.

Bitdefender. Security implications of speculatively executing segmentation related instructions on intel cpus. https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-INTEL-CPUs.pdf.

Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., and Yarom, Y. (2019). Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. (2016). Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13.

Horn, J. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html. Accessado em: 03/01/2022.

Intel. Intel software security guidance. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/feature-documentation.html.

Kemerlis, V. P., Portokalidis, G., and Keromytis, A. D. (2012). kGuard: Lightweight kernel protection against Return-to-User attacks. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 459–474, Bellevue, WA. USENIX Association.

Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

Koruyeh, E. M., Shirani, P., Khalid, A., and Abu-Ghazaleh, N. (2018). Spectre returns! Speculative execution using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD. USENIX Association.

Linux. The linux kernel documentation: Spectre side channels. https://docs.kernel.org/admin-guide/hw-vuln/spectre.html.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*.

Luyi Li, H. Y. and Tullsen, D. (2024). Indirector: High-precision branch target injection attacks exploiting the indirect branch predictor. In *33rd USENIX Security Symposium (USENIX Security 24)*.

Mambretti, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., and Kurmus, A. (2019). Two methods for exploiting speculative control flow hijacks. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA. USENIX Association.

Microsoft. *Mitigating speculative execution side channel hardware vulnerabilities* . https://www.microsoft.com/en-us/msrc/blog/2018/03/mitigating-speculative-execution-side-channel-hardware-vulnerabilities.

Oliveira, J. and Branco, R. (2023). Ret2aslr - leaking aslr from return instructions. https://github.com/google/security-research/tree/master/pocs/cpus/ret2aslr.

Ragab, H., Milburn, A., Razavi, K., Bos, H., and Giuffrida, C. (2021). Crosstalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867.

Schwarz, M., Schwarzl, M., Lipp, M., Masters, J., and Gruss, D. (2019). Netspectre: Read arbitrary memory over the network. In *European Symposium on Research in Computer Security (ESORICS)*, pages 98–115. Springer.

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

van Schaik, S., Milburn, A., Österlund, S., Frigo, Pietro Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105.

Wikner, J. and Razavi, K. (2022). RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, Boston, MA. USENIX Association.

Yarom, Y. and Falkner, K. (2014). FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA. USENIX Association.