ARTIFACT EVALUATED IEEE S&P AVAILABLE

ARTIFACT EVALUATED IEEE S&P FUNCTIONAL

ARTIFACT EVALUATED IEEE S&P REPRODUCED

VMScape: Exposing and Exploiting Incomplete Branch Predictor Isolation in Cloud Environments

Jean-Claude Graf [†]
ETH Zurich

Sandro Rüegge [†]
ETH Zurich

Ali Hajiabadi
ETH Zurich

Kaveh Razavi
ETH Zurich

[†]Equal contribution joint first authors

***Abstract*—Virtualization is a cornerstone of modern cloud infrastructures, providing the required isolation to customers. This isolation, however, is threatened by speculative execution attacks which the CPU vendors attempt to mitigate by extending the isolation to the branch predictor state. Our systematic analysis shows that this extension unfortunately is incomplete: while the most obvious case of the guest controlling branch prediction in the host has been addressed by existing hardware mitigations, we discover a number of new Spectre *Branch Target Injection* (Spectre-BTI) attack primitives on AMD Zen 1-5 and Intel Coffee Lake CPUs that, among others, enable a malicious guest to control indirect branch prediction in the host when it is executing in userspace. Using the aforementioned primitive, we craft VMScape, the first Spectre-BTI attack that enables a malicious KVM guest to leak arbitrary memory from an unmodified QEMU process running on an AMD Zen 5 server system at the speed of 154 B/s, exposing cryptographic keys for disk encryption and decryption. Our analysis of possible mitigation strategies shows that it is possible to mitigate VMScape by selectively flushing the branch predictor with minimal performance impact in common scenarios.**

1. Introduction

Spectre v2 or *Branch Target Injection* (BTI) [1] attacks have received tremendous attention recently, typically leaking privileged memory from an unprivileged user [2], [3], [4], [5], [6], [7], [8]. While these attacks show-case the violation of security boundaries, they have limited real-world impact since they assume local code execution on a user's system. The more interesting threat model is in the cloud where a malicious *Virtual Machine* (VM) can exploit Spectre-BTI to leak information from the host or another VM. The only published attacks in these scenarios, however, need attacker-controlled code in the host [1], [5], [8] which is an unrealistic assumption. Through a systematic analysis of branch prediction state isolation across virtualization boundaries, we discover a number of novel attack primitives based on Spectre-BTI. We then leverage one of these primitives to build the first Spectre-BTI exploit that leaks information from a host that is running unmodified software in default configuration.

Spectre-BTI in the cloud. Virtualization, in the form of VMs, is the primary mechanism for securely isolating co-located workloads in the cloud, protecting both customer data and the underlying infrastructure [9]. Spectre attacks, and in particular Spectre-BTI, can compromise this isolation by abusing the shared branch predictor state inside the CPU [1]. The commonly considered threat models in this scenario are a malicious VM attacking the hypervisor or another VM. As such, the hardware and software vendors give particular attention to mitigating such threats by tagging branch prediction entries learned in the VM and the hypervisor differently and flushing the branch predictor state when switching between different VMs. The residual attack surface through confused-deputy attacks on the hypervisor is difficult to exploit in practice without assuming code changes [5]. The question we investigate in this paper is if there are other unexplored attack primitives that are possible from a malicious VM.

New Spectre-BTI primitives for the cloud. We make a key observation that the existing branch predictor isolation mechanisms are too coarse-grained—they consider the hypervisor and VMs as monolithic entities while in reality they contain their own privilege levels. To investigate how existing branch predictor isolation mechanisms handle these privilege levels in different virtualization domains, we systematically analyze all possible combinations of attacker and victim protection domains in x86 CPUs, namely *Host Supervisor* (HS), *Host User* (HU), *Guest Supervisor* (GS), and *Guest User* (GU). Our analysis demonstrates that, despite existing hardware mitigations, all AMD Zen processors (Zen 1 to Zen 5) and Intel Coffee Lake are vulnerable to new *Virtualization-based Spectre-BTI* (vBTI) attack primitives between guest users and host users which we refer to as vBTI$_{GU\rightarrow HU}$ and vBTI$_{HU\rightarrow GU}$. vBTI$_{GU\rightarrow HU}$ enables new Spectre-BTI attacks on user processes running on the host from a malicious VM, and vBTI$_{HU\rightarrow GU}$ enables new Spectre-BTI attacks in scenarios where the host is assumed to be malicious and the guest has deployed hardware-assisted isolation, such as *Secure Encrypted Virtualization - Secure Nested Paging* (SEV-SNP) [10] or *Trust Domain Extensions* (TDX) [11]. We additionally discover other vBTI primitives that require consideration for other attack scenarios that we will discuss in this paper.

VMScape. To demonstrate the practicality and severity of our vBTI primitives, we present VMScape, the first Spectre-based end-to-end exploit in which a malicious guest user can leak arbitrary, sensitive information from the hypervisor in the host domain, without requiring any code modifications and in default configuration. We target the widely used *Kernel Virtual Machine* (KVM)/QEMU [12], [13] as the hypervisor, and particularly QEMU as the user-space component of the hypervisor in the host. While the vBTI$_{\text{GU}\rightarrow\text{HU}}$ primitive demonstrates the feasibility of the exploit, we address several additional challenges to enable reliable end-to-end attacks on AMD Zen 4 and Zen 5 CPUs. For instance, achieving arbitrary memory leakage requires a sufficiently large speculation window. However, obtaining a large speculation window is challenging for a guest user, since it cannot simply flush the victim branch pointer due to lack of access to physical memory or code modifications in QEMU. To overcome this, we reverse engineer AMD Zen 4 and Zen 5's cache hierarchy to build eviction sets for their non-inclusive *Last Level Cache* (LLC), thereby enabling a sufficiently large speculation window for VMScape to succeed with the gadgets that we have found in QEMU.

VMScape can leak the memory of the QEMU process at a rate of 154 B/s on AMD Zen 5. We use VMScape to find the location of secret data and leak it, all within 102 s, extracting the cryptographic key used for disk encryption/decryption as an example. Our analysis shows that mitigating VMScape requires an explicit flush of the *Branch Prediction Unit* (BPU) state in certain conditions. In particular, an *Indirect Branch Prediction Barrier* (IBPB) is necessary on each `VMEXIT` before entering the userspace hypervisor. Our evaluation shows that such a mitigation, as developed by the Linux kernel maintainers as a response to VMScape, introduces a marginal performance overhead in common scenarios.

Contributions. We make the following contributions:

- The first systematic analysis of branch predictor isolation across all possible virtualization and privilege boundaries, leading to the discovery of new primitives that are effective even with recent CPU mitigations.
- The first guest-to-host Spectre-BTI attack on unmodified software, called VMScape, using our newly-discovered vBTI$_{\text{GU}\rightarrow\text{HU}}$ primitive and new insights on cache evictions on AMD Zen 4 and Zen 5 for increasing the speculation window.
- An analysis of mitigation possibilities on various microarchitectures, leading to IBPB-on-`VMEXIT` as a basis for mitigating VMScape.

Responsible disclosure. We have disclosed VMScape to AMD and Intel PSIRT on June 7, 2025, and the issue remained under embargo till September 11, 2025. Linux maintainers mitigated VMScape with patches based on our IBPB-on-`VMEXIT` recommendation. We verified that these patches are effective in stopping our vBTI primitives. VMScape is tracked under CVE-2025-40300. Further information, including the source code of our experiments and exploit can be found at https://comsec.ethz.ch/vmscape.

2. Background

We provide some background on the *Kernel Virtual Machine* (KVM)-based virtualization stack in Linux (Section 2.1), branch prediction mechanisms (Section 2.2), Spectre-BTI attacks (Section 2.3), and their respective mitigations (Section 2.4). We conclude by motivating the need to explore Spectre-BTI attacks in the cloud scenario (Section 2.5).

2.1. KVM-Based Virtualization

Hardware virtualization extensions such as AMD SVM [10] and Intel VT-x [14] provide hardware support for running multiple operating systems concurrently on the same processor, offering strong isolation guarantees with minimal performance overhead. The *hypervisor* presents each *guest Virtual Machine* (VM) with the illusion of full control over a physical system. On Linux, KVM acts as the hypervisor and is implemented as a kernel module running on the *host*. KVM relies on a user-space component responsible for managing the lifecycle and emulated devices of a VM. We refer to this user-space process as *User-Mode Component of the Hypervisor* (Hypervisor$_{\text{user}}$). QEMU [13] is a widely-used Hypervisor$_{\text{user}}$ in Linux systems [15]. It performs tasks such as creating, starting, stopping, and migrating VMs, and emulates required hardware devices. Each guest *virtual CPU* (vCPU) maps to a dedicated thread in the Hypervisor$_{\text{user}}$, allowing the host kernel to schedule guest execution just like normal host processes since threads and processes are treated similarly by the scheduler in Linux. Consequently, the executions of the guest threads are interleaved with other guest threads and host processes.

Entering and exiting VMs. A vCPU thread either runs in "guest-mode" or "host-mode". To transition from host-mode to guest-mode, it issues a `KVM_RUN` *ioctl* to the KVM kernel module, passing a pointer to its vCPU. Depending on the platform, KVM either issues the `VMLAUNCH`/`VMRESUME` (on Intel) or `VMRUN` (on AMD) instruction. The CPU restores its state from the vCPU and starts the guest execution. When the guest executes a sensitive or privileged instruction, the CPU intercepts it, resulting in a `VMEXIT`. The CPU saves its internal state back into the vCPU structure and continues execution in KVM. Depending on the `VMEXIT` type, KVM either handles the exit directly or delegates it to the Hypervisor$_{\text{user}}$ through a `KVM_EXIT`.

Protection domains. To ensure confidentiality and integrity, computing systems must enforce isolation between entities operating in different protection domains. In many contexts, the focus is on the distinction between the *user* and *supervisor* domains (i.e. ring 0 vs ring 3). However, when considering virtualized environments, this abstraction must be extended to account for guest execution. As illustrated in Figure 1, the user domain then comprises both *Host User* (HU) and *Guest User* (GU) domains, while the supervisor domain encompasses *Host Supervisor* (HS) and *Guest Supervisor* (GS) domains.

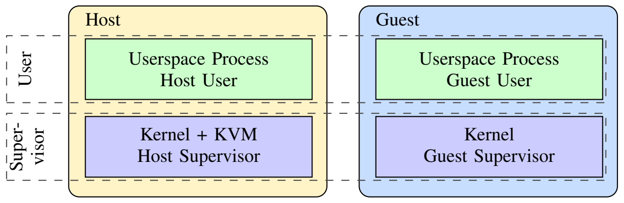


Figure 1. On x86-64 systems with VMs, protection domains include HU, HS, GU, and GS, extending beyond user and supervisor modes.

2.2. Branch Prediction

A cornerstone optimization of CPUs is the accurate prediction of branch instructions and their targets. The *Branch Prediction Unit* (BPU) in modern CPUs is highly complex, composed of different predictors tailored to different branch types and contexts. While *static predictors* only consider the source address of a branch, *dynamic predictors* use the path history of a branch as additional context [2], [3], [6], [7], [16], [17]. *Indirect branches* impose a control flow transfer to a destination either given by a register or through memory. This allows them to have a multitude of destinations that the BPU tries to correlate with the path history of the branch. Intel CPUs use a *Branch Target Buffer* (BTB) for static predictions and an *Indirect Branch Predictor* (IBP) for dynamic predictions. When available, dynamic predictions take priority over static predictions [8], [17]. On AMD CPUs, the dynamic predictor is called *Indirect Target Array* (ITA), and it is used when a branch has encountered multiple targets. Otherwise only the BTB is used [18]. Path history is recorded in the *Branch History Buffer* (BHB) on both Intel and AMD. Figure 2 illustrates the structures involved when predicting indirect branches.

2.3. Spectre-BTI

Spectre Branch Target Injection or Spectre-BTI [19] is a class of speculative execution attacks that allow an attacker to leak data across protection domain boundaries by exploiting indirect branch mispredictions. The attacker causes the BPU to predict the *victim branch* (as part of a *speculation gadget*) to a *disclosure gadget*, inducing *transient execution*. In the disclosure gadget, a secret is accessed in a way that lets the attacker later infer it through a *side channel*, typically a cache-based FLUSH+RELOAD [20]. To cause a misprediction of the victim branch, the attacker *trains* the BPU using a *training branch* that *collides* with the victim branch in the targeted predictor. To target a specific predictor, the attacker may also need to mimic the path history of the victim branch in the training branch.

Most prior work on Spectre-BTI has focused on scenarios in which an unprivileged host user process targets the host supervisor [1], [2], [3], [5], [6], [21]. In contrast, attacks between user processes have received relatively little attention [7].

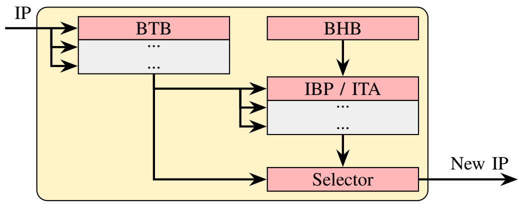


Figure 2. BPU components involved in predicting indirect branches. A BTB provides static predictions, while the IBP/ITA correlate targets with path history stored in the BHB on Intel/AMD.

2.4. Spectre-BTI Mitigations

Spectre-*Branch Target Injection* (BTI) attacks require the attacker to have control over the BTB state that is used to predict the victim, implying the need for shared BPU state between the attacker and victim domains. *Domain isolation* techniques aim to mitigate BTI attacks by preventing the sharing of BPU state across critical protection domain boundaries. *Indirect Branch Restricted Speculation* (IBRS) [22] prevents branches of lower privileged domains (e.g., user) from influencing branches of higher privileged domains (e.g., supervisor), thereby defending the kernel from user-mode attackers. Intel and AMD have deployed *Enhanced IBRS* (eIBRS) [22] and *Automatic IBRS* (AutoIBRS) [23] to improve IBRS; both mitigations are *always-on* and do not require expensive model specific register writes on privilege transitions. Mitigations that *sanitize* the BPU remove potentially malicious entries before they can influence a victim in a different protection domain. The *Indirect Branch Prediction Barrier* (IBPB) allows developers to explicitly flush BPU state during domain transitions that are not otherwise sufficiently isolated. BPU state may also be shared among *sibling threads* on CPUs with *Simultaneous Multithreading* (SMT) enabled. *Single Threaded Indirect Branch Predictor* (STIBP) [24] is a mitigation that restricts BPU state sharing across SMT threads.

Available mitigations vary across microarchitectures, and the default mitigation depends on both hardware support and kernel configuration. Table 1 shows the available mitigations on the systems we evaluated and indicates whether they are enabled by default on Ubuntu 24.04.

2.5. Motivation

While Spectre-BTI attacks from user to kernel showcase a security violation, they require code execution on the victim machine which limits the impact of such attacks. Arguably, the more interesting threat model for Spectre-BTI is in the cloud where it is easy for an attacker to rent a VM and execute their attack against the hypervisor or other VMs. Previous work [1], [5] has explored the case where a malicious guest targets the hypervisor. These attacks, however, rely on modifying the code in the hypervisor to inject the desirable gadgets, which significantly limits their practicality and impact. This raises an important question: *can Spectre-BTI attacks be mounted across virtualization boundaries without imposing overly strong assumptions?*

TABLE 1. Evaluated microarchitectures and their isolation mechanisms against Spectre-BTI.

Vendor	CPU	Year	Codename	Microarchitecture	Microcode	Isolation Mechanism			
						IBPB	IBRS	eIBRS/AutoIBRS	STIBP
Intel	Core i7-8700	2017	Coffee Lake S	Coffee Lake	0xfa	◓	●	○	◓
	Core i7-13700K	2022	Raptor Lake	Raptor Cove	0x12c	◓	○	●	●
				Gracemont	0x12c	◓	○	●	●
AMD	Ryzen 5 1600X	2017	Summit Ridge	Zen 1	0x8001137	◓	●	○	○
	EPYC 7252	2019	Rome	Zen 2	0x8301038	◓	●	○	●
	EPYC 7413	2021	Milan	Zen 3	0xa0011d3	◓	●	○	●
	Ryzen 7 7700X	2022	Raphael	Zen 4	0xa601209	◓	◓	●	●
	Ryzen 5 9600X	2024	Granite Ridge	Zen 5	0xb404023	◓	◓	●	●

○ Not available ◓ Available ● Available and Default (Ubuntu 24.04)

3. Threat Models

The goal of the attacker is to infer secrets across the host-guest virtualization boundary. In particular, we consider the following three threat models: First, $\mathrm{TM}_{G \rightarrow H}$ considers the scenario where an attacker controlling a guest VM targets the host system. Second, $\mathrm{TM}_{H \rightarrow G}$ represents a malicious host targeting a guest running on the system. This threat model is particularly relevant when considering guests that rely on Intel's *Trust Domain Extensions* (TDX) or AMD's *Secure Encrypted Virtualization - Secure Nested Paging* (SEV-SNP) to defend against malicious hosts. Third, instead of targeting the host, a malicious guest can target another guest, which is reflected by the $\mathrm{TM}_{G_1 \rightarrow G_2}$ model. We assume the use of hardware virtualization extensions: SVM [10] for AMD and VT-x [14] for Intel. Furthermore, we assume the absence of software vulnerabilities in all components, in particular in the host and guest kernel, and the $\mathrm{Hypervisor}_{\mathrm{user}}$. Finally, we assume that the systems employ all the default security mitigations against Spectre-BTI, as listed in Table 1.

4. Overview of Challenges

Existing Spectre-BTI mitigations aim to isolate BPU state across protection domain boundaries. While the most obvious case of a malicious guest attacking the hypervisor has been explored in the past [1], [5], a systematic analysis of BPU state isolation in virtualized environments when considering the different privilege levels in the guest and host is still lacking. This brings us to our first challenge:

Challenge C1. Identifying gaps in BPU's protection domain isolation in virtualized environments.

To address this challenge, we need to carefully assess the isolation of the individual structures of the BPU. In Section 5, we systematically test the isolation that is provided by the BTB and IBP/ITA when considering different privilege levels in the guest and host. Our systematic analysis shows that many of the recent microarchitectures are susceptible to some *Virtualization-based Spectre-BTI* (vBTI) primitives, indicating a lack of proper host-guest isolation in the BPU.

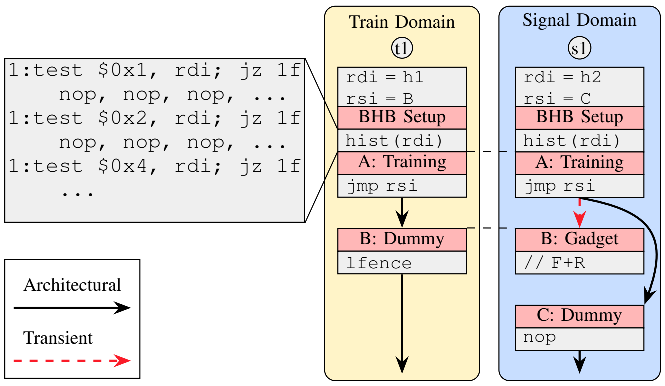
However, the exact consequences and potentials for exploitation using these new vBTI primitives are not directly evident. Our next challenge is to identify the precise circumstances under which an attacker could build relevant attacks in different threat models.

Challenge C2. Understanding and identifying different threat models affected by the porous BTB isolation in a virtualized environment.

Addressing this challenge necessitates a detailed analysis of the requirements for an attacker to exploit the identified vBTI primitives. In Section 6, we present five novel attack scenarios within the three threat models introduced in Section 3. These attack scenarios allow an adversary to attack a victim across the virtualization boundary using the identified primitives. Among these scenarios, we target a novel cross-privilege vBTI primitive for exploitation, in which a malicious guest attacks the $\mathrm{Hypervisor}_{\mathrm{user}}$ to leak sensitive information, such as cloud infrastructure secrets. Although the underlying speculation primitive is conceptually simple, constructing a practical, end-to-end exploit remains a challenge.

Challenge C3. Enabling practical exploitation and data leakage across virtualization boundaries.

The limited interaction between the guest and host software layers makes it challenging to find suitable speculation gadgets and reliable side channels. A critical requirement for successful data leakage is ensuring a sufficiently large speculation window. However, effective cache eviction on modern AMD Zen microarchitectures, in particular on Zen 4 and Zen 5, is an open problem. In Section 7, we detail the construction of the first reliable cache evictions on AMD Zen 4 and Zen 5. Building on these findings, Section 8 develops VMScape, a practical Spectre exploit that enables a guest to leak host memory across the virtualization boundary using one of our newly discovered vBTI primitives.



```
1:test $0x1, rdi; jz 1f
    nop, nop, nop, ...
1:test $0x2, rdi; jz 1f
    nop, nop, nop, ...
1:test $0x4, rdi; jz 1f
    ...
```

Figure 3. Training ((t1)) and signaling ((s1)) traces of the vBTI experiment allow us to evaluate BPU state isolation across the training–signaling domain boundary. The indirect branch source A and destination B are mapped to the same virtual addresses in both domains. Each indirect branch is preceded by a BHB setup, a sequence of branches that initializes the BHB to the same state in both traces using a seed value in `rdi`. This ensures that the BPU cannot distinguish between training and signaling branches, as both collide in the BTB and ITA/IBP. If isolation is lacking, the signaling branch deviates from the architectural target given by `rsi` and transiently executes the trained target at B. We detect this misprediction using a disclosure gadget.

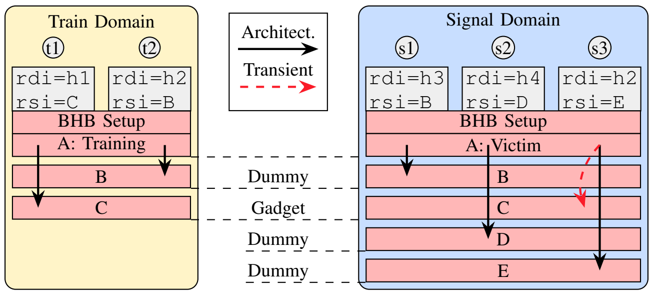


Figure 4. By combining different training traces ((t1), (t2), (s1), (s2)) we can induce the BPU to use the prediction for the signaling trace (s3) from either the BTB or the IBP/ITA. The virtual addresses of the branch source A and targets B and C are identical across both domains.

5. BPU Isolation under Virtualization

We analyze the isolation of indirect branch prediction across protection domain boundaries in virtualized environments on a range of microarchitectures. To assess the overall isolation guarantees, we need to consider the isolation of the static and dynamic indirect branch predictors individually, and test for interference among all four fundamental protection domains of x86 with hardware virtualization. These are namely HU, HS, GU, and GS. We list all systems that we evaluate in Table 1. Recent Intel CPUs like Raptor Lake feature a heterogeneous design consisting of two microarchitectures, one optimized for high-performance, the other for low-power tasks.

5.1. Methodology

To evaluate cross-domain BPU interference for indirect branches, we use a classic Spectre-BTI [1] experiment, as illustrated in Figure 3. We train in one domain, and signal for interference in another domain, using a FLUSH+RELOAD side channel [20].

This analysis requires us to execute arbitrary code in all four evaluated domains. For HS code execution, we implement a kernel module that enables a user to run arbitrary user code as supervisor. This necessitates disabling *Supervisor Mode Execution Prevention* (SMEP) and *Supervisor Mode Access Prevention* (SMAP) on the host system. For guest execution, we rely on the KVM selftest infrastructure that is part of the Linux kernel. This simplistic hypervisor allows us to map code to arbitrary guest virtual addresses and execute it. However, KVM selftests run as GS by default, leading us to extend the framework and implement an additional privilege transition to reach GU.

It is critical for our evaluation that we verify the isolation for all structures involved in the prediction of indirect branches. These structures include the BTB and IBP on Intel, and BTB and ITA on AMD. The reverse engineering requires us to control the state of the BHB to cause or prevent collision in the IBP/ITA. As shown in Figure 3, we initialize the BHB by executing a pseudo-random sequence of branches, depending on a control seed. This history setup snippet is placed directly preceding the training and victim branches, giving control over the BHB states at those branches.

Note that all modifications mentioned in this section are solely for the purpose of the reverse engineering. The final attack presented in Section 8 does not require any software modifications, and runs with the default mitigations, as listed in Table 1.

BTB prediction. As visualized in Figure 2, multiple structures are involved in the prediction of the indirect branches. Hence, a precise experiment setup is required to ensure that the prediction is served from the BTB and not the IBP/ITA. Intel always checks for a hit in the BTB but gives precedence to IBP hits whenever available [17]. AMD uses the BTB to predict all indirect branches that have only seen a single target, otherwise it prefers predictions from the ITA [18].

In the training domain, the execution of the training branch from A to B inserts an entry into the BTB on both Intel and AMD (Figure 3 (t1)). After switching to the signaling domain, we execute the signaling branch at A to C ((s1)). As A is the same virtual address in both the training and signaling domain, the BTB is unable to distinguish the two. Additionally, the distinct BHB states achieved by using seed values $h_1 \neq h_2$ ensure that the IBP on Intel cannot provide a prediction and falls back to the BTB. On AMD, since branch source A has so far only encountered the target C, no entry has been created in the ITA, making it get predicted by the BTB too. To ensure a clean state, we place an IBPB in-between each iteration. Getting a signal in this experiment shows susceptibility to vBTI.

IBP/ITA prediction. The key challenge of testing the dynamic predictor is that we need to ensure the predictions are not served from the BTB. This requires us to execute specific

TABLE 2. SUMMARY OF THE OBSERVED vBTI PRIMITIVES FOR ALL EVALUATED SYSTEMS. EMPTY HALVE CIRCLES INDICATE VULNERABLE, WHILE COLORED ONES INDICATE THE DEFAULT MITIGATION PREVENTING A LEAK.

	vBTI / vBTI-SMT Primitive									
Microarch.	HU→GU	HU→GS	HS→GU	HS→GS	GU→HU	GU→HS	GS→HU	GS→HS	G_1U→G_2U	G_1S→G_2U
Zen 1	○○	🟠🟠	○○	🟠🟠	○○	🟠🟠	○○	🟠🟠	🟣○	🟣○
Zen 2	○🟢	🟠🟢	○🟢	🟠🟢	○🟢	🟠🟢	○🟢	🟠🟢	🟣🟢	🟣🟢
Zen 3	○🟢	🟠🟢	○🟢	🟠🟢	○🟢	🟠🟢	○🟢	🟠🟢	🟣🟢	🟣🟢
Zen 4	○🟢	🔵🟢	○🟢	🔵🟢	○🟢	🔵🟢	○🟢	🔵🟢	🟣🟢	🟣🟢
Zen 5	○🟢	🔵🟢	🔵🟢	🔵🟢	○🟢	🔵🟢	🔵🟢	🔵🟢	🟣🟢	🟣🟢
Coffee Lake	○○	🟠🟠	○○	🟠🟠	○○	🟠🟠	○○	🟠🟠	🟣○	🟣○
Raptor Cove	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🟣🟢	🟣🟢
Gracemont	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🔵🟢	🟣🟢	🟣🟢

same-thread is: ○ vulnerable, or protected by: 🟠 retpoline, 🟣 IBPB, 🔵 AutoIBRS/eIBRS
cross-SMT is: ○ vulnerable, or protected by: 🟠 retpoline, 🟢 STIBP

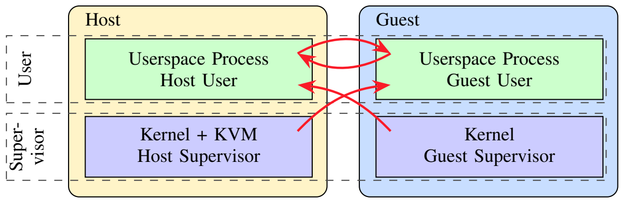
training traces, as visualized in Figure 4. In the training domain we execute the training branch at A twice, each time to a different target. To force the BPU to distinguish the two invocations, we set different history seeds $h_1 \neq h_2$. Consequently, not only an entry in the BTB is created, but also inside the IBP/ITA.

The transition from the training to the signaling domain may invalidate or isolate BTB entries. However, on both Intel and AMD, a valid entry in the BTB is necessary to get a prediction from the IBP/ITA. Therefore, we need to ensure a corresponding BTB entry is present after switching to the signaling domain. On Intel, a single encounter of the indirect branch at A (s1) is sufficient to make subsequent encounters (s3) get predicted by the IBP. On AMD, we need an additional branch encounter (s2) to a different target, before the prediction for the final branch encounter (s3) is served from the ITA. If we get a signal on a domain transition in this experiment but not to the prior one, it indicates an imbalanced isolation of the BTB and IBP/ITA.

5.2. Primitive Evaluation

To assess the isolation guarantees, we systematically test all combinations of protection domains across the virtualization and privilege boundaries. We enable all the default mitigation mechanisms listed in Table 1. Additionally, we have repeated all the experiments once with indirect jumps and once with indirect calls as the training and signaling instructions, allowing us to assess whether both are predicted similarly and isolated properly. Our results indicate no difference between indirect jumps and indirect calls, suggesting that they are handled equivalently by the mitigations. Hence, we do not distinguish between them in the remainder of this work. Moreover, we observe that the interference signal is symmetric between the guest and the host, indicating that the isolation mechanism is equivalent in both directions. We use the notation vBTI$_{\mathrm{X}\rightarrow\mathrm{Y}}$ for a vBTI primitive with training in domain X and signaling in domain Y.

Table 2 shows the susceptibility of each microarchitecture to the vBTI primitives. Since we find no differences in vulnerability between the BTB and the IBP/ITA, Table 2 does not need to distinguish between them. We observe that all microarchitectures, except for Intel Raptor Cove and Gracemont, are susceptible to some vBTI primitives. Most notably, Zen 1 through Zen 5 are vulnerable to vBTI$_{\mathrm{HU}\rightarrow\mathrm{GU}}$ and vBTI$_{\mathrm{GU}\rightarrow\mathrm{HU}}$, with Zen 1 to Zen 4 additionally vulnerable to vBTI$_{\mathrm{HS}\rightarrow\mathrm{GU}}$ and vBTI$_{\mathrm{GS}\rightarrow\mathrm{HU}}$, as visualized in Figure 5. While both Raptor Cove and Gracemont are adequately protected, Coffee Lake is also vulnerable to the aforementioned primitives for Zen 1 to Zen 4. However, we find that no microarchitecture is susceptible to any primitive targeting the HS and GS domains.


Host
Guest
User
Super-visor
Userspace Process Host User
Userspace Process Guest User
Kernel + KVM Host Supervisor
Kernel Guest Supervisor
Figure 5. Effective protection domain isolation on Zen 4, susceptible to vBTI$_{\mathrm{GU}\rightarrow\mathrm{HU}}$, vBTI$_{\mathrm{GS}\rightarrow\mathrm{HU}}$, vBTI$_{\mathrm{HU}\rightarrow\mathrm{GU}}$, and vBTI$_{\mathrm{HS}\rightarrow\mathrm{GU}}$.

Observation O1. Despite the latest mitigations being enabled, all AMD Zen microarchitectures and Intel Coffee Lake are vulnerable to vBTI$_{\mathrm{GU}\rightarrow\mathrm{HU}}$ and vBTI$_{\mathrm{HU}\rightarrow\mathrm{GU}}$. Additionally, Zen 1 through Zen 4 and Coffee Lake are affected by vBTI$_{\mathrm{GS}\rightarrow\mathrm{HU}}$ and vBTI$_{\mathrm{HS}\rightarrow\mathrm{GU}}$.

BPU isolation mechanism. AMD Zen 4 and Zen 5 employ AutoIBRS as a Spectre-BTI mitigation. While in supervisor mode, it prevents the speculative execution of indirect branch targets until they are resolved [5], [6]. By rerunning the previous experiments with AutoIBRS disabled, we can gain further insights into the isolation enforcement on Zen 4 and Zen 5 microarchitectures. While Zen 4 becomes susceptible to all vBTI primitives in the absence of AutoIBRS, Zen 5 remains unaffected by primitives that cross user–supervisor boundaries. This suggests that AMD Zen 5 CPUs feature an additional native isolation mechanism on top of AutoIBRS, equivalent to a single-bit privilege level tag for prediction entries.

Observation O2. On Zen 5, branch prediction entries are tagged with a single-bit privilege level to isolate user and supervisor domains.

However, a single bit is insufficient in distinguishing between the four domains present in virtualized environments. Consequently, AutoIBRS is still required to isolate, for example, the HS and GS domains.

Observation O3. Despite the addition of isolation bits in the BTB, the BPU on Zen 5 is unable to distinguish between host and guest domains and therefore still relies on AutoIBRS to protect the supervisor domains.

SMT isolation. To complement the previous experiments that tested same-thread isolation, we test for vBTI-SMT primitives using a modified experiment. We have a training process that iteratively executes a training branch (Figure 3 (t1)). The signaling process is then bound to the SMT sibling core and executes the signaling branch with different history ((t2)). After the signaling, we flush the BPU using an IBPB to make it *forget* about the gadget destination. As listed in Table 2, while STIBP on Zen 2 through Zen 5 and Raptor Lake successfully isolates sibling threads, the lack of STIBP on Zen 1 enables the vBTI-SMT$_{HU\rightarrow GU}$, vBTI-SMT$_{GU\rightarrow HU}$, vBTI-SMT$_{HS\rightarrow GU}$, and vBTI-SMT$_{GS\rightarrow HU}$ primitives. Furthermore, we find that although Coffee Lake supports STIBP, it is not always enabled by default, thereby allowing the same cross-SMT primitives.

Observation O4. With STIBP entirely absent on Zen 1 and only conditionally enabled on Coffee Lake, both microarchitectures appear vulnerable to vBTI-SMT$_{GU\rightarrow HU}$, vBTI-SMT$_{HU\rightarrow GU}$, vBTI-SMT$_{GS\rightarrow HU}$, and vBTI-SMT$_{HS\rightarrow GU}$.

6. Exploitation in Different Threat Models

Our experiments have shown that some virtualization boundaries are not adequately isolated, allowing branch target interference across these boundaries. In this section, we contextualize our findings with respect to the three threat models introduced in Section 3, and discuss five realistic, previously unexplored scenarios in which these findings lead to concrete security violations. We represent each attack scenario as $A \rightarrow B \mid V$, where A and B denote the protection domains of the attacker and the victim, respectively, and V refers to the specific target (e.g., a host process).

6.1. Attacks from Guest to Host

The first threat model we analyze is TM$_{G\rightarrow H}$, which considers a malicious guest targeting the host system. We identify three scenarios within this model.

(S1) **G→H | Host Process.** In this scenario, a malicious guest targets a host user process to leak a secret, like a password hash of a SUID process [7]. The victim process may either be scheduled on the same hardware thread after the guest is preempted (i.e., *same-thread*), or on an SMT sibling thread on the same physical core (i.e., *cross-SMT thread*). In the same-thread case, in addition to the lack of guest-to-host BTB isolation, the attack also requires the absence of BTB sanitization during task switches. Based on our reverse engineering observations, we find that Zen 1 through Zen 5, as well as Intel Coffee Lake, lack sufficient isolation and are vulnerable to the vBTI$_{GS\rightarrow HU}$ and vBTI$_{GU\rightarrow HU}$ primitives. For cross-SMT attacks, guest-to-host BTB interference must be coupled with shared predictor states between sibling threads, indicated by the vBTI-SMT$_{GS\rightarrow HU}$ and vBTI-SMT$_{GU\rightarrow HU}$ primitives. This condition is met on Zen 1 and Coffee Lake microarchitectures.

While these attacks are theoretically feasible, they face several practical challenges. A guest-based attacker typically has limited influence over host user processes, making it difficult to reliably manipulate or interact with them. Establishing a robust side channel between a guest and an arbitrary host user process is particularly challenging, as the attacker lacks both spatial and temporal control. In particular, the attacker cannot dictate when or where the target process is scheduled for execution, further complicating precise exploitation.

(S2) **G→H | Hypervisor$_{user}$.** This scenario considers a malicious guest targeting its own Hypervisor$_{user}$. This requires insufficient BTB sanitization on `VMEXIT`s and `KVM_EXIT`s, alongside a lack of BTB state isolation between the guest and the host on the same-thread case. Compared to general G→H | Host Process attacks, exploitation here is simpler. The guest can deliberately trigger `VMEXIT`s that are handled by its Hypervisor$_{user}$, enabling frequent attacker–victim interactions in the same-SMT case. The cross-SMT case is also more practical. Ordinarily, scheduling mitigates cross-SMT BTI attacks by reducing the likelihood of colocating attacker and victim. Here, however, the victim executes the Hypervisor$_{user}$ of the same guest, making colocation much more likely. Although the attacker can only access memory mapped into the Hypervisor$_{user}$'s address space, this can still include sensitive data, such as cloud infrastructure secrets used for networking or storage [25]. We observe that Zen 1 through Zen 5 and Coffee Lake are vulnerable to this scenario, with Zen 1 and Coffee Lake additionally being vulnerable to the cross-SMT case.

Observation O5. Cross-SMT G→H | Hypervisor$_{user}$ attacks are facilitated as the attacker and victim belong to the same guest execution, making colocation far more likely.

(S3) **G→H | Guest Kernel.** In this scenario, the attacker leverages the Hypervisor$_{user}$ as a confused deputy to extract kernel memory from the attacker's own guest instance. As such, this scenario assumes that the attacker is an unprivileged process running in the guest. It is a variant of the G→H | Hypervisor$_{user}$ scenario, but instead of leaking a Hypervisor$_{user}$ secret, it leaks the guest's own memory, that is being mapped in the Hypervisor$_{user}$. Besides the requirement for the Hypervisor$_{user}$ to have the guest's memory mapped, the prerequisites are identical to G→H | Hypervisor$_{user}$. Specifically, these prerequisites include the lack of guest-to-host BTB isolation and inadequate sanitization on `VMEXIT`s and `KVM_EXIT`s.

Observation O6. Multiple recent microarchitectures, including Zen 5, are vulnerable to G→H | Guest Kernel attacks.

TABLE 3. ATTACK SCENARIOS BY THREAT MODELS, INDICATING REQUIREMENTS, TARGET, AND VULNERABLE MICROARCHITECTURES.

				Exploitable on							
Scenario	**Target**	**SMT**	**Required Primitives**	Zen 1	Zen 2	Zen 3	Zen 4	Zen 5	Coffee Lake	Raptor Cove	Gracemont
S1 $G \to H$ | Host Process	Host user process	Same	$vBTI_{GS\to HU}$ or $vBTI_{GU\to HU}$	✓	✓	✓	✓	✓	✓	✗	✗
		Cross	$vBTI\text{-}SMT_{GS\to HU}$ or $vBTI\text{-}SMT_{GU\to HU}$	✓	✗	✗	✗	✗	✓	✗	✗
S2 $G \to H$ | $Hypervisor_{user}$	$Hypervisor_{user}$	Same	$vBTI_{GS\to HU}$ or $vBTI_{GU\to HU}$	✓	✓	✓	✓	✓	✓	✗	✗
		Cross	$vBTI\text{-}SMT_{GS\to HU}$ or $vBTI\text{-}SMT_{GU\to HU}$	✓	✗	✗	✗	✗	✓	✗	✗
S3 $G \to H$ | Guest Kernel	Guest supervisor	Same	$vBTI_{GU\to HU}$	✓	✓	✓	✓	✓	✓	✗	✗
		Cross	$vBTI\text{-}SMT_{GU\to HU}$	✓	✗	✗	✗	✗	✓	✗	✗
S4 $H \to G$ | SEV-SNP Process	Guest user process	Same	$vBTI_{HU\to GU}$ or $vBTI_{HS\to GU}$	-	-	✓	?	✗	-	✗	✗
		Cross	$vBTI\text{-}SMT_{HU\to GU}$ or $vBTI\text{-}SMT_{HS\to GU}$	-	-	✗	✗	✗	-	✗	✗
S5 $G_1 \to G_2$ | Guest Process	Other guest user process	Same	$vBTI_{G_1U\to G_2U}$	✗	✗	✗	✗	✗	✗	✗	✗
		Cross	$vBTI\text{-}SMT_{G_1U\to G_2U}$	✓	✗	✗	✗	✗	✓	✗	✗

✓ Vulnerable ✗ Not vulnerable - Not applicable ? Not supported on the tested system

6.2. Attacks from Host to Guest

The second threat model we analyze is $TM_{H\to G}$ in which a malicious host targets a guest.

S4 $H \to G$ | SEV-SNP Process. In this scenario, a malicious host targets a guest, typically one that assumes a hostile host and employs a hardware-based isolation technology such as SEV-SNP [10] or TDX [11]. Unlike in $TM_{G\to H}$, the attacker controls the scheduling of the guest thread, providing the attacker more control over the victim. Since these technologies impose additional security measures, the presence of working $vBTI_{HS\to GU}$, $vBTI_{HU\to GU}$, $vBTI\text{-}SMT_{HS\to GU}$, and $vBTI\text{-}SMT_{HU\to GU}$ primitives on systems without SEV-SNP/TDX is not sufficient to determine vulnerability of different microarchitectures. However, we were able to confirm that on Zen 3, by default, SEV-SNP does not isolate the BTB between the guest and the host. On Zen 5, we noticed that the predictions appear properly isolated between the host and an SEV-SNP guest. Unfortunately, we do not possess a Zen 4 server to check this isolation. While Zen 1 and Zen 2 do not support SEV-SNP, they could be vulnerable in scenarios using weaker technologies like SEV or SEV-ES.

Observation O7. AMD Zen 3 is vulnerable to $H \to G$ | SEV-SNP attacks.

SEV-SNP provides various optional protection mechanisms for guests, which we will discuss in Section 9.3. While our analysis demonstrates the susceptibility of older, pre-eIBRS Intel microarchitectures to $vBTI_{HS\to GU}$, $vBTI_{HU\to GU}$, $vBTI\text{-}SMT_{HS\to GU}$, and $vBTI\text{-}SMT_{HU\to GU}$ primitives, these systems do not support TDX.

6.3. Attacks Between Guests

The third threat model we analyze is $TM_{G_1\to G_2}$. In this model, a malicious guest attacks another guest running on the same host system.

S5 $G_1 \to G_2$ | Guest Process. In this scenario, a malicious guest targets a user process of another guest, highlighting the risk of co-residency between untrusted VMs. This requires a lack of BTB isolation between different guests. As with earlier scenarios, we differentiate between a same-thread and cross-SMT case. In the same-thread case, the victim executes on the same thread as the attacker, following a preemption of the attacker's thread. This additionally requires the lack of BTB sanitization on VM switches. However, this is mitigated in the Linux kernel through an IBPB on VM switches. The cross-SMT case, instead, requires that BTB state be shared across sibling threads. Based on our reverse engineering in Section 5.2, we find that both Zen 1 and Coffee Lake lack SMT thread isolation, rendering them vulnerable to $G_1 \to G_2$ | Guest Process attacks.

Observation O8. AMD Zen 1 and Intel Coffee Lake are vulnerable to $G_1 \to G_2$ | Guest Process attacks.

Practical exploitation, however, remains challenging. There is generally no direct channel for interactions between an attacker and guest VM and shared memory for a side channel on public cloud hosts.

6.4. Summary

Table 3 provides an overview of the attack scenarios, summarizing their prerequisites, intended targets, and the vulnerable microarchitectures. Our analysis demonstrates that vBTI primitives can lead to concrete security violations across all three threat models introduced in Section 3. These include attacks from a malicious guest against the host ($TM_{G\to H}$), from a malicious host against a guest ($TM_{H\to G}$), and even between two guests ($TM_{G_1\to G_2}$).

To demonstrate the feasibility of vBTI attacks, we present VMSCAPE, an end-to-end exploit targeting the $G \to H$ | $Hypervisor_{user}$ scenario. This scenario represents

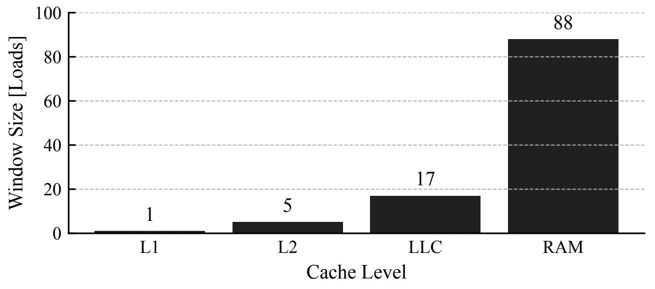


Figure 6. Speculation window size on Zen 4 as the maximum number of executed dependent loads with the pointer to the indirect branch destination residing in the respective cache level.

a particularly severe case for cloud providers, as leaking Hypervisor$_{\text{user}}$ memory targets their infrastructure.

7. Increasing the Speculation Window

Our aim is to build an end-to-end memory leak exploit in the G→H | Hypervisor$_{\text{user}}$ scenario, targeting up-to-date AMD Zen 4 and Zen 5 systems. While building this exploit, we observed that the speculation window of the victim branch was too short. We first explain how we assess the speculation window size and then discuss how we can enlarge it using cache eviction.

Speculation Window Size. To measure the size of the speculation window, we design the following experiment: we use an indirect branch that takes its target address from memory, resulting in the BPU predicting and speculating to the target. The size of the speculation window depends on how quickly the target address can resolve architecturally, which in turn depends on the cache level holding the target address of the indirect branch. We can measure the size of the speculation window by chaining n dependent loads during speculation and checking for the cache hit of the last load. We ensure that all loads, except for the last one, reside in the L1 cache. We repeat the experiment 4096 times for each $n \in [1, 127]$, and vary the cache level holding the target of the indirect branch.

As shown in Figure 6, when the target address resides in L1, only a single dependent load is executed during the speculation window. Such a speculation window is insufficient for a FLUSH+RELOAD-based side channel, since it requires at least two dependant loads. While our results for L2 indicate a sufficiently large speculation window for a perfect speculation gadget, attacks sometimes require even larger speculation windows. This can be due to additional contention from previous instructions that slow down the disclosure gadget, or due to the victim software starting to load the target address already earlier in the instruction stream (e.g., to check if it is a valid pointer.). Such effects reduce the speculation time available to the disclosure gadget, requiring the attacker to extend the overall speculation window size.

The speculation window can be extended by evicting the memory that holds the destination of the indirection branch from the cache, thereby slowing the resolution of the victim branch [1]. However, an attacker inside a VM cannot simply flush the target pointer as the physical memory that backs it is allocated to the host process and not available to the VM. Hence, the attacker needs to evict the target pointer from the cache hierarchy instead. We note that eviction should target only the necessary cache sets to avoid slowing down the entire victim execution during speculation.

TABLE 4. CACHE DETAILS ON AMD ZEN 3, ZEN 4 AND ZEN 5. THE SIZES MAY VARY BETWEEN PROCESSORS OF THE SAME GENERATION.

Core	**L1**			**L2**			***Last Level Cache* (LLC)**		
	Size	Sets	Ways	Size	Sets	Ways	Size	Sets	Ways
Zen 3	32KiB	64	8	512KiB	2048	8	32MiB	32768	16
Zen 4	32KiB	64	8	1MiB	2048	8	32MiB	32768	16
Zen 5	48KiB	64	12	1MiB	1024	16	32MiB	32768	16

7.1. LLC eviction on AMD Zen 4 and Zen 5

Recent work by Wang et al. [26] demonstrated the feasibility of creating LLC eviction sets on AMD Zen 3. We build on their insights to present the first LLC evictions for AMD Zen 4 and Zen 5. To achieve this goal, we progress through all levels of the cache hierarchy, identifying differences to Zen 3 and learning to evict progressively larger cache levels, which results in increasingly longer speculation windows. We then show how the eviction set building can be ported inside a VM and how the XOR-based indexing can be exploited to make the process highly efficient and reliable. Table 4 lists relevant cache geometries for the evaluated processors. In the interest of brevity, we first write exclusively about Zen 4 before summarizing the differences to Zen 5.

For reverse engineering purposes, we use 1 GB paging to ensure matching bits between virtual and physical addresses. We further use AMD's accurate `APERF` timer, which is accessible through the `rdpru` instruction [27]. By default, both the 1 GB paging as well as the precise counter are not available to guests. Consequently, we only use these features during reverse engineering and not for the end-to-end exploit in Section 8.

Cache access latencies. We first calibrate the memory access timing function for each cache level. This allows us to determine whether a memory access was served from L1, L2, LLC, or main memory. To determine the latencies, we proceed as follows. We start with a memory region consisting of N cache lines. We access all cache lines in pseudorandom order, recording the access times. We repeat this procedure for increasing values of N. In Figure 7, we plot the median access time for increasing memory region size N. The latency plateaus indicate the L1, L2, LLC, and RAM access latencies.

L1 eviction sets. Based on our earlier cache access measurements, we can confirm that the L1 indexing scheme on Zen 4 matches Zen 3. We can evict any L1 entry by accessing 8 distinct cache lines (corresponding to the L1 ways), all of which share matching bits $[\mathtt{6}:\mathtt{11}]$.

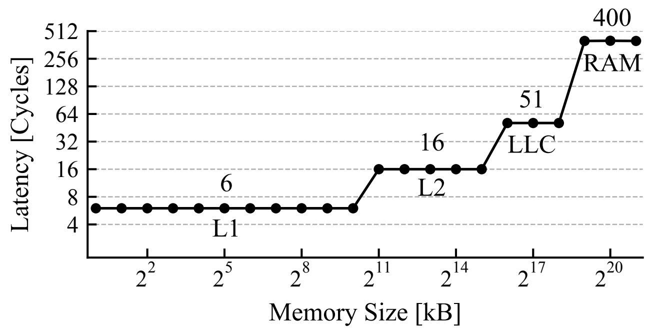
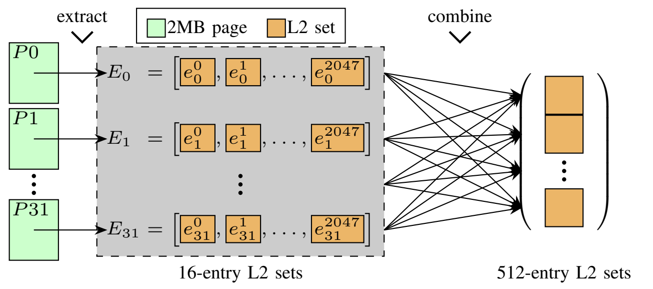


Figure 7. Median memory access latencies for a memory region of increasing size on Zen 4, shown on logarithmic axes. The latency readings of the plateaus indicate the L1, L2, LLC, and RAM latencies.



extract
2MB page
L2 set
combine
$P0$
$P1$
$P31$
$E_0 = [e_0^0, e_0^1, \ldots, e_0^{2047}]$
$E_1 = [e_1^0, e_1^1, \ldots, e_1^{2047}]$
$E_{31} = [e_{31}^0, e_{31}^1, \ldots, e_{31}^{2047}]$
16-entry L2 sets
512-entry L2 sets

Figure 8. LLC cache eviction set building strategy for VM-based attackers. In a first step, we *extract* the L2 eviction sets contained in different 2 MB pages. In a second step, we then *combine* L2 sets originating from different 2 MB pages.

L2 eviction sets. Let us assume that L2 indexing is designed in a way such that the memory inside a 2 MB huge page is evenly distributed across all L2 sets. Consequently, at least 11 indexing bits below the 2 MB page boundary are required to map to all 2048 L2 cache sets. Based on prior work on cache reverse engineering of Zen 3 [26], we further assume that some higher bits below the 2 MB page boundary are not used for L2 indexing. Given that there are 21 address bits in a 2 MB page where 11 bits are used for indexing and 6 bits are used for the offset within a 64 B cache line, we hypothesize that the 4 highest bits are not used for cache indexing. This would result in 16 entries per L2 set which we expect to be sufficient for eviction of all 8 ways.

We verify the above hypothesis as follows. We select a random 2 MB huge page and split all cache line-aligned addresses within the huge page into sets according to the value of bits [6 : 16]. We then confirm that each of the generated sets is self-evicting by measuring the mean latency of repeatedly accessing all entries within a set. We further confirm that no two generated sets are mutually evicting by measuring the latency of repeatedly accessing half the addresses from each set. Given our results, we conclude that the L2 cache indexing on AMD Zen 4 does not use bits [17 : 20].

Observation O9. Bits [17 : 20] remain unused in L2 cache indexing on AMD Zen 4.

The L2 on AMD Zen 4 has 8 ways which leads us to expect a minimal eviction set size of 8 addresses, given a *Least Recently Used* (LRU) replacement policy. However, we find that at least 12 eviction addresses are required to reliably evict a victim address from the L2 cache in our experiments. We attribute this behavior to a more complex replacement policy than LRU.

LLC eviction sets. The LLC on AMD Zen 4 is a non-inclusive victim cache for the L2. Hence, to insert cache lines into the LLC for eviction, said cache lines need to first be evicted from the L2. Given an L2 eviction set, we can identify all addresses within the same 1 GB huge page that map to this eviction set by testing whether they are evicted by it. Assuming an even distribution of addresses across the 2048 L2 sets within a 1 GB huge page, we would expect 8192 addresses to map to the same L2 set, which our experiments confirm. We have further confirmed that the first 512 of these addresses already provide reliable LLC eviction for any address within the same L2 cache set. While this does not constitute a minimal eviction set, it is sufficiently efficient for our attack and significantly increases the length of the speculation window, as shown in Figure 6 (RAM).

Zen 5. Due to the additional ways in the L1 and L2 caches, the cache eviction set sizes increase correspondingly. Zen 5 reduced the number of L2 sets compared to Zen 4 to again match Zen 3. Hence, we find that the same bits within the 2 MB pages remain unused as in Zen 3, namely [16 : 20]. Lastly, we require twice the number of addresses and have to access each address twice to reliably increase the speculation window on Zen 5 when compared to Zen 4.

7.2. Eviction from a VM

Our LLC eviction enables our attack to succeed, but it is using the precise counter and 1 GB paging which are not available to QEMU guests by default. Furthermore, increasing the size of each L2 eviction set by brute-forcing the set membership of new addresses is slow and susceptible to noise. First, we overcome the timer limitation by measuring the access times of all entries in an eviction set together rather than separately to amplify the signal. Second, we propose a significantly more efficient eviction set generation approach without relying on 1 GB huge pages.

We note that Linux, by default, aims to back VM pages with transparent 2 MB huge pages to improve performance. Hence, our approach for constructing L2 sets using 2 MB pages remains applicable from within a VM. Moreover, we argue that given many such 2 MB pages, we can *extract* L2 sets for each of them as shown in Figure 8. While this leaves us with sufficient addresses for LLC eviction, we need to find an efficient approach to *combine* L2 sets generated from different pages.

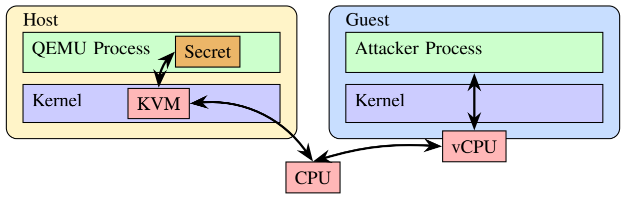


Figure 9. KVM-based guest running on a CPU with hardware virtualization support, managed by QEMU.

Let e_j^i be the i-th eviction set originating from page j where i is given by the lower L2 set indexing bits [6 : 16]. Let $E_j = \{e_j^i | i \in [0, 2047]\}$ be the set of eviction sets for page j. Our aim is to merge all L2 sets in E_j ($j \neq 0$) into L2 sets of E_0. Now, assume that for some j, a, and b, the set e_j^a collides with the set e_0^b. We would like to efficiently determine with which set e_j^c will collide. We hypothesize that the L2 and LLC are using XOR-based indexing functions which would allow calculating the colliding set e_0^d as follows:

$$d = a \oplus b \oplus c$$

Using this approach, finding the matching L2 set for a single element of each E_j ($j \neq 0$) is sufficient to determine the colliding eviction sets for all other elements. The previous approach on Zen 3 [26] would increase the size of the initial L2 sets by sampling addresses one by one and testing *each* for eviction. Our new approach only requires this process *once*, using the whole first L2 set within a 2 MB page, which then allows all other 32752 addresses from the same page to be assigned their respective set without further testing. Additionally, the approach is highly reliable, creating LLC eviction sets with 100% success rate on the first try over 100 runs whereas previous work [26] reported around 60% on Zen 3 when using 1 GB pages.

Observation O10. XOR-based cache indexing functions enable efficient merging of L2 sets from different 2 MB pages into bigger L2 sets.

With the successful extension of the speculation window, we now proceed to construct VMSCAPE.

8. VMSCAPE

Our cross-domain experiments introduced in Section 5 unveiled that all AMD Zen CPUs and Intel Coffee Lake CPUs are vulnerable to the vBTI$_{\text{GU}\rightarrow\text{HU}}$ primitive. In this section, we present VMSCAPE, our end-to-end exploit to leak hypervisor secrets as a malicious guest. Figure 9 illustrates the involved components. We target QEMU, a common Hypervisor$_{\text{user}}$ in the Linux ecosystem used in industry [15]. However, none of our assumptions in the threat model (see Section 3) or in the design of the exploit fundamentally prevent targeting a different Hypervisor$_{\text{user}}$ with a slightly modified version of the same technique. We run the attack on an AMD Zen 5 server CPU with Linux kernel version 6.8.0-85-generic and Ubuntu 25.10 packages, featuring the recent QEMU v10.0.2. We further verified that VMSCAPE leaks information successfully on an AMD Zen 4 system.

In Section 7, we addressed the speculation window challenge of the attack. However, to develop an end-to-end exploit, two additional challenges must be addressed:

- First, we need to identify a suitable exploit chain consisting of a side channel, a speculation gadget and a disclosure gadget (Section 8.1).
- Second, we need to derandomize the *Address Space Layout Randomization* (ASLR) of QEMU to find the victim branch address as well as the reload buffer address (Section 8.2).

Finally, we combine our solutions with the speculation window lengthening technique to construct an arbitrary memory leak attack against QEMU (Section 8.3).

8.1. Exploit chain

We aim to find an exploit chain in unmodified code of QEMU. To construct such a chain, we need to find a suitable side channel as well as a speculation gadget and a disclosure gadget in the QEMU binary.

Side channel. We begin with defining the side channel to be used for leaking host secrets. Through our investigation, we realized that QEMU maps all guest memory into its virtual address space. Shared memory between host and guest enables FLUSH+RELOAD as a side channel [20].

Observation O11. QEMU maps all guest memory into the Hypervisor$_{\text{user}}$ virtual address space, facilitating FLUSH+RELOAD.

FLUSH+RELOAD leaks secret information by detecting what memory locations (shared with the victim) were recently accessed. The shared memory that serves as the side channel is called a *reload buffer*.

Speculation gadget. Our exploit chain needs to trigger speculation for which our vBTI primitives use indirect calls or indirect jumps. We search for such speculation gadgets in places where we have some degree of control over registers in order to influence execution at the speculated target. Using FLUSH+RELOAD, we aim to control two registers, one containing a secret pointer and another containing the reload buffer address. Furthermore, a desired speculation gadget can be triggered repeatedly, reliably, and with high frequency, enabling high bandwidth leakage.

QEMU primarily handles I/O and device emulation when managing a KVM-based VM. Consequently, the active code footprint of QEMU is relatively small. We restrict our gadget search to general I/O handling code and avoid targeting specific device emulations, so as not to impose additional assumptions on the threat model.

We identified a suitable victim branch in QEMU's *Memory Mapped I/O* (MMIO) write handling as shown

```
 1 static MemTxResult memory_region_write_accessor(
 2                       MemoryRegion *mr,
 3                       hwaddr addr,
 4                       uint64_t *value,
 5                       unsigned size,
 6                       signed shift,
 7                       uint64_t mask,
 8                       MemTxAttrs attrs) {
 9     uint64_t tmp = memory_region_shift_write_access(
10                     value, shift, mask);
11     // [...]
12     mr->ops->write(mr->opaque, addr, tmp, size);
13     return MEMTX_OK;
14 }
```

Listing 1. Extract from QEMU's `system/memory.c` source file, showing the victim branch on line 12, with the attacker-controlled value in `tmp`.

in Listing 1. The attacker has control over the value in `tmp` which corresponds to the value written in the MMIO operation. While the adversary only directly controls a single 64 bit register (*RDX*) at the function call, dynamic analysis revealed that the same value is also left over from earlier use in a second register (*R12*). Furthermore, we can pass additional values through the shared memory, at the cost of requiring an extra load in the disclosure gadget.

Observation O12. MMIO provides repeatable, reliable, and high-frequency interaction with register control between the guest and the hypervisor.

One caveat is that while general MMIO supports 8 B writes, the device to which we write must also support writes at this granularity. For our main exploit, we use the `hpet` high precision event timer that is present in QEMU VMs by default.

Disclosure gadget. To leak secrets from the guest, we require a disclosure gadget that leverages FLUSH+RELOAD. We build on the disclosure gadget scanner from Wikner and Razavi [2]. Since the original scanner does not find the gadgets we require in the QEMU binary, we expand the scanner by improving the tainting logic for memory instructions that use differently tainted base and index registers. Additionally, we add the capability to search for chained gadgets where we combine two separate gadgets, the first of which ends in an indirect call that we can train to mispredict to the second gadget. These changes enable us to find the desired disclosure gadget. Listing 2 shows the final gadget chain used in our attack.

The gadget does not apply any multiplier to the secret which can be problematic for FLUSH+RELOAD. However, previous work demonstrated how we can rely on a technique of shifting the base of the reload buffer and checking when the secret ends up in a different cache line to overcome this limitation [2].

Branch collisions. When the victim branch in QEMU is reached, the control flow has just passed through a `VMEXIT` followed by a `KVM_EXIT`. Thus, the branch history is influenced by branches in the guest, KVM, and QEMU,

```
1 // first part of the gadget chain
2 0xac334e mov rdi, qword ptr [r12 + 0x2b58]
3 0xac3356 call qword ptr [r12 + 0x2b50]
4
5 // second part of the gadget chain
6 0x8260b5 add cl, byte ptr [rdi]
7 0x8260b7 test dword ptr [rdx + rcx], esp
```

Listing 2. Disclosure gadget in two parts. Line 2 loads the secret pointer, line 6 retrieves one byte of the secret and line 7 accesses the reload buffer.

depending on the size of the history buffer. Since AMD's history-based branch predictor takes precedence over the static BTB predictions, this necessitates either overwriting or invalidating the history-based predictions for our injected targets to be observed. Accurately replicating the history during training is possible but challenging. According to AMD's documentation [18], [28], history is only considered for indirect branches that have encountered multiple targets. By flushing the BPU using an IBPB before training, the malicious guest can ensure that the victim branch has only observed the training target. Hence, the branch is mispredicted without requiring matching history beyond the last two branches, which are relatively easy to replicate. While guests having access to IBPB may appear to be a strong assumption, this is in fact expected for guests to be able to enforce domain isolation themselves.

Observation O13. Guest software can exploit access to mitigation controls to influence what branch prediction entries are available to host software.

However, Zen 5 and later microcode updates on AMD Zen 4 modify IBPB to also invalidate direct branch type information in the BTB in order to help mitigate *Speculative Return Stack Overflow* (SRSO) [6], [29]. This inadvertently reduced the speculation window size for our attack, presumably because direct branches between the first victim pointer load and the victim branch would now slow down the speculative execution. Hence, we have to make the BTB forget that the victim branch is an indirect branch with multiple targets without using IBPB. We can do so on AMD Zen 4 and Zen 5 by replacing the victim BTB entry with a bogus direct branch prediction as follows. (1) We replace the training branch with a direct jump instruction, (2) we execute the training procedure, and (3) we write back the original training branch. This is in line with AMD's Jmp2Ret mitigation for the Retbleed vulnerability [30].

We can now reliably and repeatedly hijack the victim branch in QEMU from the guest, redirecting transient execution to our disclosure gadget.

8.2. Breaking ASLR

Linux randomizes the virtual addresses of userspace code, heap and stack in memory using ASLR by default. To mount a successful attack, we need to know the location of the speculation gadget and the disclosure gadget as well as the virtual address at which QEMU mapped the VM memory where our reload buffer resides.

Partial code ASLR derandomization. Linux implements ASLR as a random offset from the non-ASLR base address of a program. The maximum offset is a configurable parameter that defaults to some dynamic value, chosen by Linux. The target system uses a default maximum offset of 2^{32}.

Similar to prior work, we aim to find the address of the speculation gadget by checking all possible locations of the victim branch [8], [31]. They train a branch B_i at some hypothesized victim location V_i in the attacker context, trigger the execution of the victim and measure whether the prediction for B_i was evicted. This approach is not applicable in our case for two main reasons. First, in contrast to Intel, the BTB on AMD Zen 4 and Zen 5 does not overwrite the target with every misprediction but rather inserts the new target into the ITA while keeping the previous target in the BTB. Second, we would need to separately check up to around 2^{32} locations, which takes prohibitively long.

We optimize the above approach by reversing the attacker and victim roles as proposed by prior work [7]. Given a hypothesized victim branch location V_i, we can calculate the corresponding target address T_i. Thus, we can first trigger the victim to train the branch predictor and then we execute a matching branch in the attacker domain at location V_i to some non-signaling target. By additionally placing a signaling gadget in the attacker domain at the hypothesized victim target location T_i, we can detect collisions. Using our new approach, a single execution of the victim branch and a single reload of the reload buffer for signaling suffice to check many potential victim locations (e.g., 1024 locations). Once a signal is observed, we can determine which of the locations was correct.

Results. While this approach provides us with single victim location on Zen 4—directly breaking code ASLR, on Zen 5 we get a multitude of candidates. Because all candidates share the same lower 24 bits, we conclude that this is due to aliasing and the fact that the BTB provides partial targets, similar to Intel [8], [17]. While this reveals the lower 24 bits of the victim branch, it only partially derandomizes code ASLR which randomizes 32 bits above the page offset bits. We find than when training the predictor with only matching lower 24 bits, it also only provides the lower 24 bits of the target. Consequently, having only partially derandomized code ASLR, we can only speculate to targets within an offset of 24 bits from the victim branch. However, our selected disclosure gadget chain resides at a larger distance (than 2^{24} from the victim branch), requiring us to fully break code ASLR.

Locating the reload buffer. Before we can proceed to fully breaking code ASLR, we need to identify the virtual address where QEMU has mapped the memory used by the attacker's VM to back the reload buffer. The reload buffer we also need to eventually leak arbitrary data through FLUSH+RELOAD. Similar to prior work [2], we brute force the location of the reload buffer. The partial derandomization from before allows us to speculate to targets residing within a 24 bit offset from the victim branch. Within this region, we locate a gadget that loads the memory at the address provided by the attacker. Conveniently, as previously observed, Linux and QEMU collaborate to map VM pages as 2 MB huge pages. Hence, by mapping the reload buffer as a 2 MB huge page inside the VM, we can significantly reduce the number of potential reload buffer locations. Additionally, we find that since searching for the reload buffer only requires a single load in the speculation window. This means that we do need to lengthen the speculation window for this step.

Results. Over 100 repetitions, we can derandomize the reload buffer location with a success rate of 100% and within median 46 s and 83 s on Zen 5 and Zen 4, respectively.

Fully breaking code ASLR. Having located the reload buffer, we can now fully derandomize code ASLR and locate the speculation getdget. This also reveals the location of the disclosure gadget, since ASLR only randomizes the base address of the executable and preserves offsets. This extra step is not required for Zen 4 as the BTB did not seem to exhibit aliasing, directly yielding the victim location.

At the victim branch, the register *RAX* contains a pointer to some code location. Knowing this value would allow us to calculate the full victim branch location. We can leak this pointer using an additional gadget that is within the 24 bit offset of the victim branch. This gadget adds the attacker controlled register *RDX* to *RAX* and loads the resulting address. We leak *RAX* by brute forcing values for *RDX*, such that we register a hit in our reload buffer.

Results. Fully breaking code ASLR takes a median 12 s and 75% accuracy on Zen 5, neglecting the time to locate the reload buffer. On Zen 4, where the first step already revealed the location of the victim branch, breaking ASLR takes 238 s and has 70% accuracy. The difference in time comes from the fact that in the first step of code ASLR breaking, we only need to find a single candidate within 2^{12} possibilities in on Zen 5, while on Zen 4, finding the correct address requires checking all 2^{32} possible locations. The larger search space for Zen 4 in the first step takes significantly longer than the required second step for Zen 5 where we leak the remaining 20 bits of ASLR entropy.

8.3. Arbitrary Memory Leak

We now have all the components required for building an arbitrary memory leak attack: the exploit chain, an ASLR-breaking primitive, and a reliable approach to provide sufficiently large speculation windows.

Our VMSCAPE attack proceeds as follows. First, we locate the victim branch using our ASLR-breaking technique. Once the gadgets are identified, we can determine the QEMU virtual address of our reload buffer. Next, we construct our LLC eviction sets and identify the correct eviction set by testing which one enables dependent loads to leave a cache trace. We then use our gadgets with the long speculation window to leak QEMU's internal data structures to find the address of the secret data and then leak said secret data.

Results. We need a median $102\,\mathrm{s}$ for the full end-to-end attack on Zen 5, leaking a $4KiB$ QEMU secret object, which we set to be a disk encryption and decryption key as an example. Of the total attack time, $58\,\mathrm{s}$ are needed to find the victim branch and the reload buffer. Once obtained, our exploit leaks arbitrary QEMU memory at a rate of $154\,\mathrm{B/s}$ with 99.85% byte accuracy. The overall success rate of the attack is 68%.

9. Discussion

We discuss NUMA considerations for the VMSCAPE attack (Section 9.1), additional vBTI variants that we discovered after our initial responsible disclosure (Section 9.2) and the mitigation strategies for vBTI primitives (Section 9.3).

9.1. NUMA considerations

We noticed that VMSCAPE cannot leak certain memory pages on our Zen 5 system that employs NUMA. Tracking this issue down, it turns out that the Linux NUMA balancing mechanism [32] periodically marks pages as not present to track whether they are most frequently accessed by CPU cores in the same NUMA node. Pages that are marked as not present cannot be accessed in speculation—effectively stopping VMSCAPE on memory pages that are not regularly accessed by the hypervisor. However, cloud virtualization platforms and providers typically allocate physical memory for VMs on the same NUMA node to improve performance [33], [34], [35]. In such practical settings, the Linux kernel recommends turning off NUMA balancing to reduce the unnecessary tracking overhead [32].

9.2. BHI variants

Intel's response to classical Spectre-BTI [1] was eIBRS, a hardware mitigation that effectively isolates user and supervisor entries in the BTB. *Branch History Injection* (BHI) [3], however, re-enabled cross-privilege Spectre attacks by exploiting the lack of BHB isolation between the user and supervisor domains. As we show in Section 5, eIBRS successfully enforces BTB isolation in virtualized environments. Yet, akin to BHI, eIBRS does not inherently prevent a guest from (partially) controlling the branch history. While this case is mitigated for attacks against the HS [5], [36], we believe that there is no mitigation for the HU. While we did not verify this, Intel has confirmed that this is indeed the case, indicating that recent Intel CPUs are potentially vulnerable to *Virtualization-based Spectre-BHI* (vBHI) primitives.

9.3. Mitigation

As described in Section 5, our reverse engineering revealed multiple vBTI primitives across recent microarchitectures. In this section, we examine mitigation strategies to prevent such primitives, beginning with IBPB-on-`VMEXIT`, which forms the basis of the deployed VMSCAPE patches.

IBPB-on-`VMEXIT`. This mitigation protects the host in the $\mathrm{TM}_{\mathrm{G}\to\mathrm{H}}$ model by flushing the BPU state with an IBPB on every `VMEXIT`. As a result, it enforces BTB isolation between guest and host on each guest-to-host transition. However, IBPB-based mitigations typically impose substantial performance overhead [2], [6], which becomes particularly problematic when applied to a fast path such as `VMEXIT`.

To quantify this overhead, we benchmarked the mitigation using the UnixBench test suite [37], following the methodology of prior work [2], [6], [8]. We run the workloads in a guest with $4\,\mathrm{GB}$ of memory and two cores, in multithreaded mode. The geometric mean of the performance overhead (over five repetitions) is $57\,\%$ on Zen 5.

Linux kernel developers based their mitigation on IBPB-on-`VMEXIT`, but optimized it by conditionally issuing the IBPB only on `KVM_EXIT`s that are followed by guest execution. Since not every `VMEXIT` results in a `KVM_EXIT` and subsequent userspace execution, this change moves the IBPB off the fast path while preserving its security guarantees. They refer to this mitigation as *"IBPB before exit to userspace"*. This patch gets applied to all affected systems, including AMD Zen 5 and recent Intel CPUs such as Lunar Lake and Granite Rapids.

In addition to verifying the effectiveness of the patches, we benchmarked their performance overhead on the same setup. UnixBench shows only a marginal overhead of $1\,\%$ on Zen 4. However, as a compute-oriented benchmark, UnixBench causes relatively few exits to userspace. To evaluate a workload that causes frequent exits to userspace, we used `fio` [38] to generate disk I/O, by randomly reading and writing $10\,\mathrm{GB}$ on a `virtio` disk repeatedly for 10 times. On Zen 4, the optimized mitigation has an overhead of $51\,\%$, approaching the $57\,\%$ overhead of IBPB-on-`KVM_EXIT`. When using devices handled by the host kernel (`vhost`) or a pass-through devices, exits to userspace are minimized. These results demonstrate that the performance impact of the mitigation strongly depends on both the workload and the QEMU configuration, but is expected to remain marginal in most real-world settings.

Retpoline. Retpoline is a classic software-based Spectre-BTI mitigation that replaces certain indirect branches (`jmp` and `call`) by other indirect branches (`ret`) that are predicted using a different predictor [39]. Compiling $\mathrm{Hypervisor}_{\mathrm{user}}$s using retpoline would protect them against attackers in the $\mathrm{G}\to\mathrm{H} \mid \mathrm{Hypervisor}_{\mathrm{user}}$ scenario on systems not also vulnerable to earlier attacks against returns [6]. While not protecting other host user processes considering $\mathrm{G}\to\mathrm{H} \mid$ Host Process, the induced performance overhead is lower compared to IBPB-on-`VMEXIT`. Performance evaluation of retpoline using the UnixBench workloads shows an overhead of $3.9\,\%$. Combining retpoline with selective host process isolation using the `PR_SET_SPECULATION_CTRL` `prctl` can serve as an alternative strategy to the high-overhead IBPB-on-`VMEXIT`, depending on the threat model. Our investigations show that none of the popular open-source Hypervisor$_{\text{user}}$s deploy retpoline [13], [40], [41], [42]. This mitigations is only applicable to CPUs that are unaffected by Phantom speculation [43], like Zen 5.

Protecing SEV-SNP Guest. While Intel systems affected by vBTI primitives do not support TDX, SVM on Zen 3 through Zen 5 provides several optional protections for an SEV-SNP-enabled guest [18], [28]. (1) *Branch Target Buffer Isolation* restricts execution outside the guest domain from influencing branches within the guest execution. (2) *Indirect Branch Prediction Barrier on Entry* forces the CPU to issue an IBPB on every `VMRUN`. (3) *SMT Protection* forces the sibling SMT thread to be idle while the guest is running on the other sibling thread. Our experiments in Section 5 indicated that Zen 5 already provides sufficient isolation for SEV-SNP guests by default. Hence, such mitigations are appropriate on earlier CPUs.

Protecting SMT. STIBP is a hardware mitigation that isolates the BPU of SMT cores. We identified that most systems enable STIBP by default, protecting against vBTI-SMT primitives. However, STIBP is not *always-on* in some CPUs, like Coffee Lake. Systems that do not support STIBP, such as Zen 1, have no option other than to disable SMT entirely, resulting in significant performance impact [2], [44].

Hardware-based isolation. Ultimately, the root cause of vBTI primitives is the lack of BPU state isolation between host and guest domains. Zen 5 CPUs introduce privilege domain tagging to distinguish between user and supervisor domains. A similar tagging to distinguish the host and the guest can prevent vBTI primitives and should be considered for future revisions of the Zen microarchitectures.

10. Related Work

There is a substantial body of research on microarchitectural security spanning decades, starting with timing side channels, in particular on caches [45], [46]. Acıiçmez et al. [47], [48] presented similar work but targeting early branch predictors. Evtyushkin et al. [31] later demonstrated that modern BTBs can be similarly exploited to break ASLR. Ge et al. [49] presented a classification of such (and many other) exploitable timing side channels through the shared resources and contexts.

Spectre. Kocher et al. [1] presented the first transient execution attack caused by branch misprediction. A host of other transient execution attacks then followed [50], [51], [52], [53], [54], [55], [56], [57]. Canella et al. [58] provide an evaluation and classification of many such attacks. We categorize prior speculative execution attacks into three groups based on their victim protection domain: (1) hypervisor attacks, (2) supervisor attacks, and (3) userspace attacks.

(1) Hypervisor attacks. The original work on Spectre [1], alongside later work [5], [8], provided proof of concept attacks against modified supervisor components of hypervisor software. Google researchers partially published an attack against KVM [59], based on Inception [6]. Our work, in contrast, presents the first real end-to-end attack where a malicious KVM guest targets *userspace* components of the hypervisor.

(2) Supervisor attacks. Supervisor mode has been an attractive target for BTI exploits early on [1]. While CPU vendors and operating system developers scrambled for mitigations, researchers continued to find gaps in the new defenses. Wikner and Razavi circumvented retpoline in Retbleed [2] and in-hardware mitigations in Phantom [43] and Inception [6] by invalidating assumptions about predictor training. BHI and follow-up work demonstrated repeatedly that same-mode training is possible despite various measures against it [3], [4], [5], breaking assumptions made by cross-privilege BTI mitigations.

(3) Userspace attacks. Various attacks against userspace have been proposed, in particular against browser sandboxing. Ragab et al. [60] generalized machine clears as a source of speculative execution and identified novel causes of machine clears that they exploited in the browser. Ret2spec [61] and Spectre Returns [21] both explore *Return Stack Buffer* (RSB) predictions to attack userspace victims in various scenarios. Spring [62] then showed that despite various system and in-browser mitigations, RSB predictions could still be exploited. More recent attacks also target other predictors to similarly leak secrets across sandbox boundaries [63], [64]. Wikner and Razavi [7] then were the first to demonstrate a full end-to-end cross-process attack. In contrast to previous attacks targeting userspace, we identified and exploited primitives that operate not only across context, but also across privilege and virtualization boundaries simultaneously.

11. Conclusion

We presented VMSCAPE, the first practical Spectre-BTI attack from a malicious KVM guest on QEMU, leaking memory at $154\,\mathrm{B/s}$ on AMD Zen 5. Unlike previous Spectre-BTI attacks from a malicious guest, VMSCAPE does not require any hypervisor code modification. VMSCAPE achieves this using a novel vBTI primitive that we discovered by systematically exploring isolation gaps in the branch predictor state between the guest and host at different privilege levels on different AMD and Intel CPUs. Mitigating VMSCAPE requires an IBPB after a `VMEXIT` when entering a userspace hypervisor, which introduces a marginal performance overhead.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their feedback. We would also like to thank the Intel PSIRT, AMD PSIRT and Linux security teams for coordination and collaboration on this issue. Furthermore, we thank Alexandra Sandulescu from Google for her feedback on our mitigation analysis. This work was supported by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1–19.

[2] J. Wikner and K. Razavi, "RETBLEED: Arbitrary Speculative Code Execution with Return Instructions," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3825–3842.

[3] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 971–988.

[4] S. Wiebing, A. d. F. Tron, H. Bos, and C. Giuffrida, "InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 577–594.

[5] S. Wiebing and C. Giuffrida, "Training Solo: On the Limitations of Domain Isolation Against Spectre-v2 Attacks," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, May 2025, pp. 3599–3616.

[6] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing New Attack Surfaces with Training in Transient Execution," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7303–7320.

[7] J. Wikner and K. Razavi, "Breaking the Barrier: Post-Barrier Spectre Attacks," in *S&P*, May 2025.

[8] S. Rüegge, J. Wikner, and K. Razavi, "Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions," in *USENIX Security*, Aug. 2025, intel Bounty Reward, BlackHat USA presentation. [Online]. Available: Paper=https://comsec.ethz.ch/wp-content/files/bprc_sec25.pdfURL=https://comsec.ethz.ch/bprc

[9] M.-M. Bazm, M. Lacoste, M. Südholt, and J.-M. Menaud, "Isolation in cloud computing infrastructures: New security challenges," *Annals of Telecommunications*, vol. 74, no. 3, pp. 197–209, Apr. 2019.

[10] Advanced Micro Devices, Inc., "AMD64 APM, Volume 2: System Programming," Tech. Rep. 24593, Mar. 2024.

[11] "Intel Trust Domain Extension," Tech. Rep.

[12] "Kernel virtual machine," https://linux-kvm.org/page/Main_Page.

[13] "QEMU: A generic and open source machine emulator and virtualizer," https://www.qemu.org//.

[14] "Intel64 SDM, Volume 3 (3A, 3B, 3C & 3D): System Programming Guide," Tech. Rep.

[15] "Proxmox: Simplify your data center," https://proxmox.com/en/.

[16] H. Yavarzadeh, A. Agarwal, M. Christman, C. Garman, D. Genkin, A. Kwong, D. Moghimi, D. Stefan, K. Taram, and D. Tullsen, "Pathfinder: High-Resolution Control-Flow Attacks Exploiting the Conditional Branch Predictor," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. La Jolla CA USA: ACM, Apr. 2024, pp. 770–784.

[17] L. Li, H. Yavarzadeh, and D. Tullsen, "Indirector: High-Precision Branch Target Injection Attacks Exploiting the Indirect Branch Predictor," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2137–2154.

[18] Advanced Micro Devices, Inc., "Software Optimization Guide for the AMD Zen5 Microarchitecture," Tech. Rep. 58455, Aug. 2024.

[19] Y. Zhang and R. Sion, "Speculative Execution Attacks and Cloud Security," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, ser. CCSW'19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, p. 201.

[20] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.

[21] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[22] "Indirect Branch Restricted Speculation," Tech. Rep.

[23] Advanced Micro Devices, Inc., "Amd64 Technology Indirect Branch Control Extension," Tech. Rep., Jul. 2018.

[24] "Single Thread Indirect Branch Predictors," Tech. Rep.

[25] "Providing secret data to QEMU," https://www.qemu.org/docs/master/system/secrets.html.

[26] H. Wang, M. Tang, Q. Wang, K. Xu, and Y. Zhang, "ZenLeak: Practical Last-Level Cache Side-Channel Attacks on AMD Zen Processors," 2025.

[27] Advanced Micro Devices, Inc., "AMD64 APM, Volume 3: General-Purpose and System Instructions," Tech. Rep. 24594, Mar. 2024.

[28] ——, "Software Optimization Guide for the AMD Zen4 Microarchitecture," Tech. Rep. 57647, Jan. 2023.

[29] "Speculative Return Stack Overflow (SRSO) — The Linux Kernel documentation," https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/srso.html.

[30] Advanced Micro Devices, Inc., "Technical Guidance for Mitigating Branch Type Confusion," Tech. Rep.

[31] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[32] Documentation for /proc/sys/kernel. The Linux Kernel documentation. [Online]. Available: https://docs.kernel.org/admin-guide/sysctl/kernel.html#numa-balancing

[33] Configure NUMA-aware VMs — Google Distributed Cloud (software only) for bare metal. Google Cloud. [Online]. Available: https://cloud.google.com/kubernetes-engine/distributed-cloud/bare-metal/docs/vm-runtime/numa

[34] How ESXi NUMA Scheduling Works. [Online]. Available: https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/8-0/vsphere-resource-management-8-0/using-numa-systems-with-esxi/how-esxi-numa-scheduling-works.html

[35] NUMA - Proxmox VE. [Online]. Available: https://pve.proxmox.com/wiki/NUMA

[36] "Branch History Injection and Intra-mode Branch Target Injection," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html.

[37] "UnixBench test suite," https://github.com/kdlucas/byte-unixbench.

[38] "Flexible I/O Tester," https://github.com/kdlucas/byte-unixbench.

[39] "Retpoline: A software construct for preventing branch-target-injection," https://support.google.com/faqs/answer/7625886.

[40] "VirtualBox: Powerful open source virtualization For personal and enterprise use," https://www.virtualbox.org/.

[41] "Firecracker: Secure and fast microVMs for serverless computing."

[42] "Crosvm: The ChromeOS Virtual Machine Monitor."

[43] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting Decoder-detectable Mispredictions," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*. Toronto ON Canada: ACM, Oct. 2023, pp. 49–61.

[44] P. Michaud and A. Seznec, "A Comprehensive Study of Dynamic Global History Branch Prediction," Report, INRIA, 2001.

[45] W.-M. Hu, "Lattice scheduling and covert channels," in *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, 1992, pp. 52–61.

[46] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[47] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Topics in Cryptology – CT-RSA 2007*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 225–242.

[48] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 312–320. [Online]. Available: https://doi.org/10.1145/1229285.1266999

[49] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," 2018.

[50] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[51] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, p. 991.

[52] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," 2018. [Online]. Available: https://arxiv.org/abs/1807.03757

[53] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," 2018. [Online]. Available: https://arxiv.org/abs/1806.07480

[54] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 7179–7193. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/moghimi

[55] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.

[56] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[57] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[58] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[59] "Google CPU security research," https://github.com/google/security-research/tree/master/pocs/cpus.

[60] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1451–1468. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ragab

[61] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2109–2122. [Online]. Available: https://doi.org/10.1145/3243734.3243761

[62] J. Wikner, C. Giuffrida, H. Bos, and K. Razavi, "Spring: Spectre returning in the browser with speculative load queuing and deep stacks," in *WOOT*, 2022.

[63] J. Kim, D. Genkin, and Y. Yarom, "Slap: Data speculation attacks via load address prediction on apple silicon," in *S&P*, 2025.

[64] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, "Flop: Breaking the apple m3 cpu via false load output predictions," in *USENIX Security*, 2025.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

In this paper, the authors further investigate the susceptibility of x86 CPUs to Spectre-BTI vulnerabilities, focusing on virtualization boundaries. The authors found several problems in the isolation enforcement across such boundaries, leaving host to guest and guest to host attack surface. The paper analyzes this attack surface for different CPU classes and demonstrates attacks on Zen4 CPUs.

A.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field
- Other

A.3. Reasons for Acceptance

1) This paper is the first to analyze Spectre-BTI across virtualization boundaries. Presented results are impactful especially in cloud servers where virtualization is ubiquitous, and the proposed attacks are realistic. CPU vendors and OS developers have confirmed the existence of the vulnerabilities and are working on patches to address them.
2) The paper proposes new techniques for LLC evictions on Zen-class CPUs that can further aid the analysis of other side-channel attacks against such CPUs.
3) The paper presents an end-to-end attack on Zen4 CPUs that are widely-adopted in cloud machines.

A.4. Noteworthy Concerns

1) While the paper presents an analysis of different attacks across virtualization boundaries and with different CPUs, attacks are only demonstrated on the Guest to Host-User scenario and AMD Zen4 CPUs. This raises some doubts in reviewers about direct applicability to other scenarios.