GhostBuster: understanding and overcoming the pitfalls of transient execution vulnerability checkers

Andrea Mambretti*†, Pasquale Convertini†, Alessandro Sorniotti†, Alexandra Sandulescu†,
Engin Kirda* and Anil Kurmus†
*Khoury College of Computer Sciences, Northeastern University, Boston, USA
Email: {mbr, ek}@ccs.neu.edu
† IBM Research - Zurich, Rueschlikon, Switzerland
Email: pasqualeconvertini95@gmail.com, {aso, asa, kur}@zurich.ibm.com

***Abstract*—Transient execution vulnerabilities require system administrators to evaluate whether their systems are vulnerable and whether available mitigations are enabled. They are aided in this task by multiple community-developed tools, transient execution vulnerability checkers. Yet, no analysis of these tools exists, in particular with respect to their shortcomings and whether they might mislead administrators into a false sense of security. In this paper, we provide the first comprehensive analysis of these tools and underpinning methodologies. We run the tools on a large set of combinations of Intel/AMD architectures and Linux kernel versions and report on their efficacy and shortcomings. We also run these tools on 17 of the most prominent cloud providers, report the collected results and present the current status on the preparedness of the IT hosting industry against this class of attacks. Finally, we present a new tool called `GhostBuster`, that combines methodologies and results gathered by existing tools to provide a more accurate view a system's stance against transient execution attacks for a given use case.**

***Keywords*—Transient Execution; Hardware Security; Side Channels; Microarchitectural Attacks; Vulnerability Checkers; Cloud Security; Operating System Security;**

I. Introduction

With the discovery of Spectre and Meltdown in 2018, many threat models required a complete revision to include the new category of transient execution attacks. These attacks leverage optimizations or bugs in the microarchitecture of modern CPUs to cross privilege boundaries and leak data [3].

In general, exploiting transient execution vulnerabilities requires satisfying two conditions: the presence of a *microarchitectural attack vector* and the fulfillment of all the constraints of the underlining *threat model*. For instance, to perform the Meltdown [20] attack, two conditions have to be met: first, the CPU has to defer the supervisor bit check until instruction retirement (microarchitectural attack vector) and second, the privileged memory region we want to access should be mapped in the same address space of the process (threat model requirement). Similarly, Spectre-PHT [17] requires speculative execution support on the target system and a double array access pattern in the victim process (microarchitectural attack vector). Moreover, it also requires a secret within the victim address space that we want to leak (threat model requirement). Another example is Spectre-BTB [17], where the attack vector is represented by the ability of a process to inject entries in the branch target buffer for the co-located thread, and the threat model requires that the victim process contains a side channel primitive allowing leakage of sensitive data. While some of these attacks are exploitable in real-world settings, others have not left the realm of theoretical vulnerabilities.

Owing to the high number of transient execution attacks (both practical and theoretical) and the multitude of mitigation mechanisms, the degree to which systems are safe against speculative execution attackers is often hard to determine. The determination is made even more complex by the fact that a system might be affected by a certain microarchitectural attack vector, while being safe in practice because the conditions of the associated threat model are not met. Furthermore, it is possible for a system to be unaffected by an attack vector under a certain threat model, while being affected under a different one: an example of this is Spectre-BTB, where most of the defense mechanisms are applied between kernel and user-space, but are not enforced when the attack is performed uniquely in user-space.

```
CVE-2017-5715 aka 'Spectre Variant 2, branch target
    injection'
* Mitigated according to the /sys interface: YES
    (Enhanced IBRS, IBPB: conditional, RSB filling)
* Mitigation 1
    * Kernel is compiled with IBRS support: YES
        * IBRS enabled and active: YES
    * Kernel is compiled with IBPB support: YES
        * IBPB enabled and active: YES
* Mitigation 2
    * Kernel has branch predictor hardening (arm): NO
    * Kernel compiled with retpoline option: YES
    * Kernel supports RSB filling: UNKNOWN
        (kernel image missing)
> Status: NOT VULNERABLE (IBRS + IBPB are mitigating the
    vulnerability)
```

Listing 1. Sample output from a commonly used tool to check vulnerability of a system against transient execution attacks

To illustrate the practical implications of this observation, we give in Listing 1 an example of the output of one of the most commonly-used tools used to analyze a system's status with respect to speculative execution attacks, `spectre-meltdown-checker`. The output of the tool seems to indicate that the system is not vulnerable to Spectre-BTB (also known as Spectre v2). In reality, though, this output only concerns attacks directed at the kernel. This also outlines the need to precisely distinguish *use cases* when assessing whether a system is vulnerable to transient execution attacks, such

as targeting the kernel and targeting privileged userspace programs.

An example of the impact that such misleading information can have is shown by Bhattacharyya et al. [2] and Mambretti et al. [23]. In both cases, the authors show that it is possible to perform branch target injection in patched systems although `spectre-meltdown-checker` reports that the system is not vulnerable. Indeed, neither attack targets the kernel, which is the use case implicitly considered by `spectre-meltdown-checker`, they both target user space applications controlled by the attacker.

Because of these considerations, tools designed to gauge the security posture of systems against transient execution attacks often give incomplete or erroneous information, either leaving systems potentially exposed to attacks that might be considered mitigated, or forcing expensive countermeasures for attacks that are only theoretical. Therefore, a gap exists for tools directed towards system administrators, analyzing whether for a given use case, their system is vulnerable to transient execution attacks.

In this paper, we answer the question: *To what extent can existing tools tell a system administrator whether a system is vulnerable to transient execution attacks?* To answer this question, we consider the tools available at the time of writing which check the security of a system in relation to transient execution attacks. Some of these tools gather system information and try to infer which attacks the system is vulnerable to. Others, instead, run proof-of-concept attacks in the system and try to assess empirically whether transient execution attack vectors are exploitable. Our results empirically show that none of the available tools are able to cover correctly the entire spectrum of attacks. Based on this analysis, we propose a meta-tool prototype, called `GhostBuster`, that more accurately informs of a system's vulnerability against transient execution attacks, in particular by taking into account the target use case and combining advantages of both information gathering and empirical tools. Along with `GhostBuster`, we provide 5 main recommendations for system administrators on how to use and improve such vulnerability checkers.

In this paper, we make the following contributions:

- We analyze transient execution vulnerability checkers, and report on identified shortcomings and pitfalls of these tools.
- We provide the first large-scale analysis for transient execution vulnerabilities on commercial cloud providers.
- We provide recommendations based on 6 different use cases to correctly test for transient execution vulnerabilities. As a result of our work, we provide a meta-tool, `GhostBuster`, that combines and enhances existing tools to avoid the pitfalls observed in the state-of-the-art.

II. Background & Related Work

In this section, we give detailed information about the attacks, defenses and tools analyzed in this work. Specifically, in Section II-A, we describe transient execution attacks; in Section II-B, we report the available defense mechanisms; finally, in Section II-C, we describe in detail the available tools we consider for this work.

A. Transient execution attacks

Transient execution attacks are commonly divided into two major families: *Speculation-based* and *Fault-based* [3]. In the *Speculation-based* family we find the various versions of the Spectre attacks that leverage speculative execution to achieve data exfiltration. In the *Fault-based* family instead, we find all the different variants of Meltdown that rely on bugs in the way the CPU handles faults to achieve similar results.

In this work, we include transient execution attacks for which the `transientfail` tool was able to identify a successful attack, at the time we started the analysis. In the speculation-based family, we focus on the following Spectre variants (following the naming introduced by Canella et al. [3]): Pattern History Table [17] (PHT), Branch Target Buffer [17] (BTB), Return Stack Buffer [18], [21] (RSB) and Store to Load [10] (STL). Concerning the fault-based family, we focus on the following Meltdown variants: Bound Range [3] (BR), Read-Write [16] (RW), User/Supervisor [20] (US), General Protection [15], [1], [14] (GP), Protection Key [3] (PK) and Foreshadow [26], [30] (P). In particular, we do not cover Meltdown-family variants that were reported later such as RIDL [28], ZombieLoad [24], LVI [27], and variants identified by SPEECHMINER [31]. This does not affect our conclusions on the adequacy of existing tools, which apply to existing variants.

Transient execution attacks can be further classified according to where the training of a predictor occurs. In particular, we define the following relevant configurations: *i*) *same address-space* (sAS) where training occurs in the same address space as that of the victim process, or *ii*) *cross address-space with simultaneous multi-threading* (cHT) where training occurs in a separate process running on the same physical core, or *iii*) *cross address-space without simultaneous multi-threading* (cAS) where training occurs between two processes running interleaved on the same physical core. Note that the cHT setting is a setting where attacker training and victim speculation occur in temporal colocation (different logical core, same time), whereas in the cAS setting they occur in spatial colocation (same logical core, different time). These settings are important because several attacks and mitigations rely on them. Thus, considerations on the specific setting must be taken into account in the threat model when evaluating the security posture of a system.

1) Spectre Family: The Spectre family of attacks leverages out-of-order execution and CPU speculation to misdirect the execution flow towards sensitive paths with attacker controlled inputs, that would not be otherwise executed. If the sensitive path contains a sequence of instructions able to leave traces in the cache based on a secret, the attacker can later retrieve such secret by probing the cache with known techniques (e.g. Prime and Probe, Flush and Reload). The various Spectre variants target CPU internal components and optimizations that can lead to such mispeculation.

Spectre-PHT, also known as Spectre v1, mistrains the Pattern History Table (PHT) used by the CPU to make a prediction on the outcome of conditional branches. This can lead the CPU to mispeculate a branch towards a sensitive piece of code (e.g. a Spectre v1 gadget).

Spectre-BTB, also known as Spectre v2, poisons the

Branch Target Buffer (BTB) to steer speculative execution to a mispredicted branch target previously injected by the attacker.

Spectre-RSB, similarly to Spectre-BTB, targets backward edge control flow transfer, rather than the forward edge. The attacker, through poisoning of the Return Stack Buffer (RSB), is able to hijack the prediction caused by a `ret` instruction when the saved instruction pointer on the stack is not immediately ready.

Spectre-STL, also known as Spectre v4, leverages a completely different source of speculation inside modern CPUs. It targets data forwards happening between the store buffer and the load buffer when an in-fly load requires a value from the same location of a previously executed store. In this case, the attacker can trick the CPU to forward to a load a wrong value that is subsequently used during speculative execution.

2) Meltdown Family: The Meltdown family of attacks exploits bugs within the CPU. During a Meltdown attack, the attacker tries speculatively to perform operations that are not allowed due to privilege boundaries. Meltdown relies on the fact that faults are handled by the CPU only when the faulty instruction retires, leaving speculative execution to continue across privilege boundaries before the fault is registered. This can allow an attacker to create a cache side channel with data retrieved from a higher privilege level through out-of-order execution. Each variant of Meltdown exploits a different fault type.

Meltdown-US is the first discovered variant of the Meltdown family. 'US' refers to the *user/supervisor* permission check to restrict access to memory pages.

Meltdown-RW exploits the read-write fault (#RW), generated when a process tries to write to a read-only page.

Meltdown-P, known as Foreshadow, specifically affects Intel CPUs. At the heart of this Meltdown variant lies the so called L1 Terminal fault (L1TF) [26], triggered when an address translation processing must be aborted due to a specific set of conditions (e.g. the present (P) bit cleared in the page table entry).

Meltdown-PK exploits the #PF fault raised when a process tries to access a page shielded with a Protection Key (PK).

Meltdown-BR exploits the Bound Range fault (#BR) raised in case of an out-of-bounds memory access.

Meltdown-GP is a variant of Meltdown that exploits the General Protection (#GP) exception, raised whenever an unprivileged process tries to access a privileged register, for example by using the `rdmsr` instruction.

B. Defenses

Several mitigations have been developed to protect against the attacks presented in the previous section. Some of these mitigations repurpose existing instructions or sequence of instructions to block speculative execution in sensitive areas of the code. Others, instead, are new hardware features that hardware vendors introduced in new iterations of the CPU. Changes are also made at the operating system level, including the re-design of entire subsystems. Hereafter, we describe the mitigations currently available at the time of this writing:

Memory Fencing. Through the application of serialization instructions, such as `lfence` on Intel, it is possible to force the CPU pipeline to wait for prior instructions to retire and, as a consequence, to block speculation and related Spectre attacks. Such a pipeline interruption is an expensive operation, therefore fencing instruction should be placed carefully either manually or at compile time only where really needed. For instance, the Linux kernel uses manually instrumented code, if the Spectre mitigations are enabled.

Branchless masking is a mitigation against Spectre-PHT attacks to harden load instructions that are gated by a condition. The technique involves introducing a *data* dependency (usually called mask) on the condition through a set of instructions which set the mask to zero in case the condition is false (and to unsigned negative one otherwise). The mask is then used to zero out pointers or array indices before performing the load when invalid. Masking is for example used in the Linux kernel [6] to block Spectre-PHT whenever a value coming from userspace is used as an index for an array access. It is also available as a compiler option (Speculative Load Hardening (SLH) [4]) to instrument conditional branches with control-flow dependent pointer masking.

Retpoline. As an answer to Spectre-BTB, the Retpoline [25] compile-time mitigation replaces indirect branches with `ret` instructions to prevent branch poisoning. This method ensures that return instructions always speculate into an endless loop through the RSB.

KPTI mitigates Meltdown-US and fortifies KASLR. KPTI is based on KAISER [3] (short for *Kernel Address Isolation to have Side-channels Efficiently Removed*). If KPTI is enabled, whenever user-space code is running, Linux ensures that only the kernel memory pages required to enter and exit syscalls, interrupts and exceptions are mapped. With no other pages mapped, KPTI prevents the use of kernel virtual addresses from user-space because they cannot be correctly translated.

IBRS [12] prevents indirect branch predictors executed in privileged code from being trained by less privileged code (i.e kernel-space cannot be influenced by user-space). This also includes the case of attacks from another logical core on the same physical core (cHT).

IBPB [11] prevents code that executes before it from affecting branch prediction for code that executes after. When enabled, an IBPB barrier runs across user mode or guest mode context switches. In this way, a different user cannot attack a process of another user running on the same machine. On Linux machines, IBPB can be conditionally or fully enabled: in the first case, the barrier is raised only when switching to processes that request it using `seccomp` or `prctl`.

STIBP [13] splits the branch predictor across sibling threads of a core, removing the attack vector constituted by a process training the predictor of a co-located victim process. This prevents attacks like Spectre-BTB in the cHT setting.

RSB filling blocks Spectre-BTB and Spectre-RSB in the cAS setting. It flushes the RSB whenever the CPU switches across privilege levels. For instance when the CPU switches from usermode to kernel mode the RSB might contain poisoned entries introduced by the attacker which might affect its

speculative control flow. The process of RSB filling removes any such entry.

SSB mitigation. When software-based mitigations are not feasible (such as inserting an `lfence` instruction between the store and the load instructions) for Spectre-STL, some CPUs support Speculate Store Bypass Disable that can be used to mitigate speculative store bypass. When SSBD is set, loads will not execute speculatively until the address of older stores are known.

PTE inversion is used to block Meltdown-P attacks. When a page table entry points to a non present page, the upper address bits are inverted so that an access to such entry resolves to uncacheable memory access. Given that Meltdown-P can only exfiltrate data from L1 cache, forcing the address translation to return an uncacheable address closes the attack vector.

VMC conditional cache flushing is adopted on virtualized environments. The mitigation flushes the L1 cache at every VMENTER instruction. This way secrets possibly stored in the cache are no longer accessible via the Foreshadow-VMM attack.

C. Testing Tools

In this work, we analyze and use the 4 state-of-the-art tools to discover if a system is vulnerable to transient execution attacks. We divide the tools into two main categories: *information gathering* and *empirical*.

1) Information gathering tools: These tools use the kernel, the `cpuid` instruction and the microcode version as sources to collect the necessary information. The kernel provides information about Spectre and Meltdown vulnerabilities through the `sysfs` virtual file system interface. They use the `cpuid` instruction to determine if the CPU supports mitigations such as IBRS. Due to the differences among the Linux distributions and the way the necessary pieces of information are presented by the several Linux kernel versions, these tools must constantly be adapted to parse information correctly on all the different distributions.

The `spectre-meltdown-checker` [19] script was released soon after the disclosure of Spectre and Meltdown. This script inspects the local system for information related to transient execution patches. The `spectre-meltdown-checker` script even understands if the kernel space is hardened against transient execution attacks by also disassembling the kernel image and counting the number of `lfence` instructions. The tool is constantly updated by the community with tests for all the newly disclosed attacks. It is considered the state-of-the-art tool for verifying the system status.

The `mdstool-cli` [29] tool was published together with the disclosure of the RIDL vulnerability to the public. It aims to detect if the system is vulnerable to such attacks and previously discovered ones. As for `spectre-meltdown-checker`, `mdstool-cli` inspects locations inside the system such as *sysfs* and the results of the *cpuid* instruction. While more recent, `mdstool-cli` is less exhaustive compared to `spectre-meltdown-checker` and only checks whether the CPU self-reports itself as vulnerable to a specific class of attacks. Mitigations status and availability information are also gathered but they are not factor in the final report.

2) Empirical tools: As the name suggests, these tools use an active approach to assess if the machine is vulnerable or not to transient execution attacks. The tools under this category run several tests to verify the presence of the attack vectors of each of the known transient execution attacks. Each test emulates a specific attack and determines whether the transient execution attack is feasible. Two methods can be employed to make the final determination: either cache side channels or Performance Monitoring Counters (PMCs).

The approach with PMCs is less noisy than the cache-based ones and provides a more accurate determination about the presence or absence of the attack vector. The main drawback of using PMCs is their limited availability on virtual machines: they are rarely virtualized.

Transient Fail was released by Canella et al. [3]. It includes a series of empirical tests/PoCs that cover all the known transient execution vulnerabilities. This tool attempts to trigger the vulnerabilities locally and determines whether the attempt is successful by observing micro architectural side effects on the cache. Based on those, it is possible to infer whether a specific attack vector is present in the tested system.

Speculator [22] leverages CPU performance counters to evaluate speculative execution. We extend `speculator` with tests for each known transient execution attack since they are not available in the original version of the tool. We use `markers` presented along with `speculator` [22] as signals to verify whether the attack was successful. This mode of operation differs from the one used in `transientfail` that relies instead on known cache side-channels (e.g. `Prime+Probe`, `Flush+Reload`). While Das et al. [7] suggest that PMCs should not be used in security applications to detect attacks, Mambretti et al. [22] prove instead that they can be reliably used to monitor tests results in the context of transient execution attacks. Moreover, `speculator` implementation carefully follows Das et al. [7] guideline to eliminate common mistakes in PMC usage.

TABLE I. CPU families the tools have been tested with the corresponding kernel version

CPU Family	Kernel
Intel Ivy Bridge	4.15.0
Intel Haswell	4.15.0
Intel Broadwell	5.0.0
Intel Skylake	4.15.0
Intel Kaby Lake	4.15.0
Intel Kaby Lake R	4.15.0
Intel Cascade Lake	4.15.0
AMD Ryzen	4.15.0
AMD Ryzen 2	4.15.0

At the time of our analysis, `spectre-meltdown-checker` and `mdstool-cli` were last updated at the end of May 2019, while `transientfail` on August 2019. Another tool, called SafeSide [9], is under development with the same goal and design of `transientfail`. However, due to the early stages of the tool when our experiments were run and its great similarity with `transientfail`, we decided not to include it in our comparison.

III. Methodology

In this section, we describe the methodology we follow to evaluate the tools. At first we describe a few meaningful contexts in which the tools may be run: we focus on 6 practical use cases in which a system administrator may want to execute one or more of the tools to measure the degree of security of their system against a speculative adversary, keeping the use case of that system into account. We also describe the systems in which we tested the various tools.

A. Use Cases

This section details the use cases we considered to evaluate the impact of transient execution attacks. We categorize use cases based on the security domain at which attacker and victim operate. The considered cases are the following.

- *Sandbox to process (S-P)*: we consider an attacker running sandboxed code, such as Javascript, eBPF, or NaCL [32], aiming to leak data from the process that runs the sandbox.
- *User to user (U-U)*: we consider an attacker controlling an unprivileged process, targeting a privileged process such as a process running as `root` on the same machine.
- *User to kernel (U-K)*: we consider a local attacker targeting an OS kernel.
- *VM Guest to Guest (G-G)*: we consider an attacker that controls a VM guest OS, and targets other VM guests running on the same host.
- *VM Guest to Host (G-H)*: we consider an attacker that controls a VM guest OS, and targets the hypervisor (also known as VM Monitor).
- *Host to SGX (H-SGX)*: we consider an attacker in control of the system (i.e., able to load its own kernel) targeting an SGX enclave. SgxPectre [5] presents a series of attacks in this setting, exploiting different variants of Spectre v2.

B. Systems and Platforms

This section describes the choice of platform, architecture and Linux kernel version that are used to evaluate the tools. We test each tool on several different CPU families and kernel versions: Table I shows the CPU families and the corresponding kernel version where the 4 tools have been tested. We run the *information gathering* tools on each system and collect their final report. The approach of this set of tools consists of simply parsing information gathered in the system, thus the execution time of both `mdstool-cli` and `spectre-meltdown-checker` is roughly constant. Concerning *empirical* tools, we execute each test available in `speculator` and `transientfail` on the machines listed in Table I. We set a 20 seconds timeout for all `transientfail` empirical test runs with the exception of the Spectre-BTB one where the threshold was set to 250 seconds because its runs are significantly slower. Most of the tests are meant to run indefinitely while others until a secret is revealed. The timeout is necessary to avoid infinite iterations. Spectre-BTB is a rather noisy attack and the test is run in batches with an incremental number of iterations (e.g. 100k, 200k, 300k etc.) until either the attack is successful or the time expired. We adjusted its timeout to allow this incremental search of the right number of iterations. For `speculator` instead, we execute all tests with 10k runs per instance. While in the case of `transientfail` tests can be run as a standard user, `speculator` requires root access on each machine to access the PMC interface.

We run all tools on systems hosted by 17 different infrastructure-as-a-service (IaaS) cloud providers: we collect samples for all three main IaaS offerings, namely:

- *multi-tenant* solutions: each tenant gets access to a virtual machine deployed on a set of physical resources shared with other customers;
- *single-tenant* solutions: the tenant gets access to a machine with a hypervisor whose physical resources are not shared with other customers;
- *bare metal* solutions: the machine is delivered without any virtualization solution and is fully devoted to a single customer.

Testing on real cloud systems as opposed to self hosted and managed hardware provides useful additional insight on the use of these tools in real-world scenarios: the choice of virtualization platform and its effects on the availability of the PMC infrastructure, the impact of workload from other tenants on the cache side channels, and default configurations for the systems are just a few. More will be discussed in Section IV-A1.

Overall we perform tests on 28 different machines as many cloud providers support more than one of the above mentioned solutions. To keep our analysis as general as possible, we collect the results from clouds of different sizes and market share: according to Gartner [8], the clouds we test, cover together more than the 75% of the IaaS market share in 2018.

IV. Results

In this section, we report results related to the comparison between the methodologies underpinning the tools, underlining pros and cons of each. We also present the results of our analysis on the 17 tested providers. Finally, we report, based on the use cases described in Section III-A, which methodology might be better suited to determine the security of a system with respect to transient execution attacks.

```
CVE-2017-5753 aka 'Spectre Variant 1, bounds check bypass'
* Mitigated according to the /sys interface:
        YES   (Mitigation: usercopy/swapgs barriers and
           __user pointer sanitization)
* Kernel has array_index_mask_nospec:
        YES   (1 occurrence(s) found of x86 64 bits
           array_index_mask_nospec())
* Kernel has the Red Hat/Ubuntu patch:
        NO
* Kernel has mask_nospec64 (arm64):
        NO
> STATUS:  NOT VULNERABLE  (Mitigation: usercopy/swapgs
    barriers and __user pointer sanitization)
```

Listing 2. `spectre-meltdown-checker` sample output for Spectre PHT

A. Tools comparison

Table II presents a comparison between the tools we employed in our study. We enumerate here advantages and disadvantages of each, providing examples from our analysis as support.

TABLE II. Major pitfalls and limitations observed for each tool. We indicate with ✓ that the pitfall is present, whereas we leave blank otherwise.

	Parsing Error	Use Case Imprecision	Attack Vector only	Cache Noise	PMC req.	Root req.
mdstool-cli	✓	✓				
spectre-meltdown-checker		✓				✓
transientfail			✓	✓		
speculator			✓		✓	✓

TABLE III. Classification of the result types for each of the attacks with respect to the use cases described in Section III-A. Empirical tools do not focus on specific use cases but rather on the existence of the attack vector. The table reflects this by referring the use case as synthetic. Results are reported as: ✓ if the tool reports information about a certain attack within the use case considered; ⇒ if the information can be inferred but it is not directly reported; ✗ where either no information can be inferred from the tool; finally the cell is left blank where the attack cannot be performed or it is not feasible under the specific use case.

		Spectre				**Meltdown**					
Tools	**Use Case**	PHT	BTB	RSB	STL	US	RW	P	PK	BR	GP
spectre-meltdown-checker	S-P	✗	✗	✗	⇒		✗		✗	✗	
	U-U	✗	✗	✗							
	U-K	✓	✓	⇒	✓	✓	✗	✓		✗	⇒
	G-G	✗	✗	✗				✓			⇒
	G-H	✗	✗	✗				✗			
	H-SGX	✗	✗	✗				✓			
mdstool-cli	S-P	✗	✗	✗	✗		✗		✗	✗	
	U-U	✗	✗	✗							
	U-K	⇒	⇒	✗	⇒	⇒	✗	⇒		✗	✗
	G-G	✗	✗	✗				✗			✗
	G-H	✗	✗	✗				✗			
	H-SGX	✗	✗	✗				✗			
transientfail	synthetic	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
speculator	synthetic	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗

1) Pitfalls: Parsing errors. Information gathering tools may wrongly parse system information, thereby providing wrong data to the user. For example, the `mdstool-cli` tool uses *sysfs* file system provided by the Linux kernel to detect which mitigations are used against Spectre-PHT. The content of this file has changed over different kernel versions, and `mdstool-cli` only recognizes output with an outdated format, which leads it to conclude that a system with mitigations for Spectre-PHT is vulnerable, in contradiction with the kernel-provided output. In contrast, `spectre-meltdown-checker` parses the output correctly for all tested kernel versions.

A similar parsing problem arises on older kernels when the **/sys** interface is not present. While `spectre-meltdown-checker` reports that the interface is missing, `mdstool-cli` does not and reports that CPUs are safe against Spectre-PHT, Spectre-BTB, Meltdown-US and Meltdown-P. This issue may mislead the tool user into considering a system safe. We discovered the flawed `mdstool-cli` reports while running our experiments on one of the cloud providers listed in Table V.

Inaccuracies from implicit assumptions. A second, very common issue for information gathering tools is related to the assumptions that are implicitly made over the considered use case: the tools report results in very generic terms, whereas in reality they only analyze a specific use case. For example, both `spectre-meltdown-checker` and `mdstool-cli` report results about Spectre-BTB, when in reality they only consider the effects of the Spectre-BTB attack vector on a user-to-kernel use case. Similar arguments can be made for most transient execution attacks, wherein the same attack vector applies to multiple use cases: the tacit assumptions and lack of precision can be misleading. A related and more subtle issue concerns the confusion between the attack vector and mitigations against specific attacks leveraging it. As an example, the Linux kernel includes software mitigations to prevent a subset of Spectre-PHT attacks; `spectre-meltdown-checker` checks whether they are enabled and – based on that – reports whether the system is vulnerable to Spectre-PHT. This report, shown in Listing 2, is however misleading because the mitigations it evaluates do not necessarily protect from all Spectre-PHT attacks. First, Spectre-PHT attacks are not limited to attacks targeting the kernel: any user-space program could be the target of such an attack, as long as the target program contains a vulnerable code pattern and has not been compiled with mitigations such as SLH [4]. Second, Spectre-PHT attacks against the Linux kernel may still be feasible in isolated cases, as the mitigations are based on false-negative prone static analysis and manual code analysis.

Not considering same-address space training. We have identified that Spectre-BTB attacks are implicitly assumed by empirical tools to be in the cross-address space (cAS or cHT) variant. This is problematic and may lead to a false sense of security: Spectre-BTB attacks may also be performed in the same address-space (sAS) setting. Mitigations such as Intel's IBRS, STIBP, and IBPB only apply to the cross address space setting.

Attack vector only. Empirical tools draw conclusions based on synthetic attack scenarios. Thus, they are only able to report results on the presence or absence of the specific transient attack vector on which an end-to-end attack may be built. However, the presence of the attack vector does not necessarily mean that an attack could be mounted in a particular use case, i.e. that the system is vulnerable. For example, in the context of Spectre-BTB in the cHT setting, an empirical tool can verify with a synthetic attack if the branch predictor is shared or not between the logic cores of a system. Although the attack is synthetic, a negative result means that the attack vector is not present, and that no attack of this kind can be performed. However, a positive result does not mean a system is vulnerable: it only means that further requirements need to be fulfilled for a valid end-to-end attack.

Cache noise. `transientfail` faces problems when the system under test has competing cache activity. This can give the user wrong or inaccurate results. During our cloud analysis, we experienced such a problem on one provider where none of the `transientfail` proof-of-concept attacks ran correctly.

PMC & root requirement. A problem specific to `speculator` is the availability of PMCs. In particular, during our analysis of cloud providers, in all but one cloud provider PMCs were not available, due to the lack of the performance counter interface in the virtualized environment. Also, `speculator` requires root privileges to use the PMCs infrastructure, which limits the use of the tool to the administrators of the system.

TABLE IV. Tools version used in the experiments

Tool	Commit hash
`spectre-meltdown-checker`	91d0699
`mdstool-cli`	11b3240
`transientfail`	7b0c9b2
`speculator`	4973a19

2) Tools vs use cases: Table III describes the effectiveness of each tool in evaluating transient execution attacks in each of the applicable use cases.

`spectre-meltdown-checker` reports information for the *U-K* use case for almost all attack families. It is also the only tool the user can run to determine if Meltdown-P is patched in three out of the six use cases, and the only tool that checks the CPU microcode to determine whether the machine is patched against Meltdown-GP. `mdstool-cli` limits its output to the information exposed by the kernel through `sysfs`; this mainly includes information about the state of the mitigations, with few details about the use cases under analysis. Nevertheless, a user with good knowledge of the mitigations listed in the final report of this tool can infer if the machine is vulnerable in the *U-K* use case.

Empirical tools verify instead the existence of the attack vectors at the base of Spectre and Meltdown variants. Both `transientfail` and `speculator` execute test applications in user space in order to verify the feasibility of an attack. However, the attacks are usually run in the same address space of the victim, or from an attacker to a victim process not hardened against the attack. Therefore, the empirical tools reports are not linked with the use cases considered for the other tools; results are thus labeled as *synthetic* in Table III.

B. Analysis

In this section, we present results collected by running all tools on systems hosted by 17 of the most prominent cloud providers.

1) Information gathering tools: Results from `spectre-meltdown-checker` and `mdstool-cli` show that 16 out of the 17 tested cloud providers make use of the proper mitigations to harden the kernel against transient execution attacks: only one provider uses a kernel version (4.4.0) that does not include mitigations against Spectre and Meltdown attacks.

Regarding Spectre, 16 cloud providers run a kernel properly recompiled with `lfence` instructions and retpoline. Additionally, all enable mitigations such as IBPB, STIBP and SSBD. The default setup of these mitigations is *conditionally enabled* which means that the mitigations are not enforced to user space applications unless specifically requested, however the patched kernel in these settings uses them when needed.

TABLE V. List of the 17 cloud provider tested and their available configurations

Cloud	Multi-tenant	Single-tenant	B. Metal
AWS	✓	✓	✓
Alibaba	✓		
Azure	✓	✓	
IBM Cloud	✓	✓	✓
GCP	✓	✓	
Digital Ocean	✓		
OVH			✓
Hetzner	✓		✓
Oracle	✓	✓	✓
Packet			✓
Scaleway	✓		✓
Vultr	✓	✓	✓
Bigstep			✓
Cloudsigma	✓		
Tencent	✓		
RamNode	✓		
Zenlayer			✓

Hence, the kernel cannot be attacked in any of the depicted scenarios (sAS, cAS and cHT).

Regarding Meltdown, all the tested Intel machines, excepting those of one provider, use KPTI and PTE inversion to block respectively Meltdown-US and Meltdown-P. However, the Meltdown-US test from the `transientfail` suite showed that KPTI does not stop Meltdown-US over pages mapped as kernel-pages at runtime. Moreover, `spectre-meltdown-checker` reports that the CPU microcode of such machines is patched to stop Meltdown-GP. As for AMD-based machines, Meltdown attacks do not affect them.

2) Empirical tools: Here, we report the results obtained by running the *empirical* tools on the considered cloud providers. Tests for Spectre-PHT and Spectre-BTB succeed in the cAS case on all machines. Not all cloud providers are affected in the cHT case since on virtual machines, hyper-threading is not always available, making this scenario unfeasible. Specifically, we find that 4 clouds disable hyper-threading in their multi-tenant solutions. Results for Spectre-RSB are negative, indicating that this attack vector is prevented owing to the fact that RSB filling is enabled: at every context switch, possibly poisonous RSB entries are flushed. The cHT scenario is unfeasible because the RSB is only shared between processes running interleaved on the same logical core. Spectre-STL succeeds on all machines since SSBD is only conditionally enabled, being inactive in practice: SSBD must be fully enabled to prevent the attack in both user-space and kernel-space. Tests for Meltdown-RW succeed on all Intel-based nodes except for those based on the new Cascade Lake microarchitecture. For what concerns Meltdown-BR, the tests are successful on Intel CPUs supporting the `mpx` instructions, while they report negative results on Ivy Bridge and Broadwell families. None of the Meltdown variants show positive results on AMD.

We conclude that the *empirical* tools generally report most known attack vectors as present and exploitable. Instead, the *information gathering* tools report the security stance of the system in the U-K use case, which generally results in the system being classified as not vulnerable due to the widely enabled kernel mitigations. The results discrepancy is justified by the following:

- the default configuration of all tested systems only conditionally enable user-space (U-U) mitigations;
- the code implementing the synthetic attack scenarios in

speculator and transientfail restricts generalization to other target applications on the system. By design, these tools are unaware of the mitigations enabled outside of the synthetic environment.

V. Recommendations

During our analysis, we find that all available tools have shortcomings about their ability to determine whether a system is vulnerable. Based on the experience gained during our work, we propose 5 main recommendations directed towards systems administrators using such tools, as well as developers of these tools.

A. Limit cache noise

When working with empirical tools such as transientfail, it is very important to pay particular attention to workloads running on the same physical machine. Heavy cache activity, for instance from the last-level-cache (LLC) that is often shared across cores might cause the test to report that an attack is not feasible while actually the failure is caused by temporary cache activity. If the user/administrator has the ability to control the load on the system, we recommend to pause any workload during the test's execution. Additionally, we recommend to run the tests several times with enough time between each run. Finally, when possible, we recommend using PMC-based approaches whenever available, as they are less prone to noise.

```
CVE-2018-3639 aka 'Variant 4, speculative store bypass'
* Mitigated according to the /sys interface: YES
   (Speculative Store Bypass disabled via prctl and seccomp)
* Kernel supports disabling speculative store bypass (SSB):
    YES (found in /proc/self/status)
* SSB mitigation is enabled and active: YES
    (per-thread through prctl)
> STATUS: NOT VULNERABLE (Mitigation: Speculative Store
    Bypass disabled via prctl and seccomp)
```

Listing 3. spectre-meltdown-checker sample output for Spectre STL

```
========= SPECTRE STL =========
 * Attack success rate (synthetic test): 93.0%
 * Attack vector: Present
 * Difficulty: High - No practical attack demonstrated

   ------ USE CASES -------
 * S-P: SSB is not fully disabled. Check that the victim
     program is compiled with seccomp()/prctl().
 * U-U: SSB is not fully disabled. Check that the victim
     program is compiled with seccomp()/prctl().
 * U-K: SSB is not fully disabled. Your kernel is vulnerable
     .
 * G-G: Check if SSB is fully disabled on the host machine.
 * G-H: Check if the kernel on the host machine supports
     disabling SSB.
 * H-SGX: SSB is not fully disabled.
```

Listing 4. GhostBuster sample output for Spectre STL

B. Define the right use case and understand your threat model

Most of the mitigations currently available (described in Section II-B) incur a medium to high overhead. In fact, it is very common for these mitigations to be disabled by default. Enabling one or more of them when not necessary is generally not indicated. Therefore, it is important for a system administrator or user to understand which of the attacks flagged by one of the tools for a specific system falls under the use case(s) the administrator/user cares about. An example of such a case would be enabling a system-wide mitigation like STIBP when the only use case considered important is the User to Kernel (U-K) one. This is because, by default, STIBP already prevents attacks such as Spectre-BTB under the U-K scenario and it would be a performance waste to enforce it system-wide.

C. Understanding information gathering tools results

Tools should report as clearly as possible the assumptions underlying the reported results and the considered use cases. In the current state of these tools, we recommend to verify either through the tool documentation (if any) or the source code of the tool (if available) which use cases is under analysis. Our analysis suggests it is possible that the assumptions made by existing tools do not match those of the user, leading to a false sense of security or a lack of action.

```
========= SPECTRE BTB same address-space =========
* Attack success rate (synthetic test): 95.00%
* Attack vector: Present
* Difficulty: High - No practical attacks demonstrated.
  ------ USE CASES -------
* S-P: Check that the target process is compiled with lfence
     or retpoline.
* U-U: Check that the target process is compiled with lfence
     or retpoline.
* U-K: This kernel is not vulnerable: Full retpoline + IBPB
    are mitigating the vulnerability.
* G-G: Check that the target process is compiled with lfence
     or retpoline.
* G-H: Check that host kernel is compiled with retpoline and
     supports RSB filling.
* H-SGX: Check that the target process is compiled with
    lfence or retpoline.
```

Listing 5. GhostBuster sample output for Spectre BTB same address-space

D. Use a mixed approach

Based on our analysis, we make the observation that the two types of tools, information gathering and empirical, report different types of information. While using only one or the other gives a very limited snapshot of the system security, their combination allows to gather more robust information about the system in general but also allows for a greater ability to infer information and inform the user. For instance, if we consider the Spectre-BTB in the cross address-space case and we successfully verify the presence of such attack vector using either speculator or transientfail. Now, we can combine this result with the output on spectre-meltdown-checker and connect which use cases this attack vector might affect. For example, assuming that spectre-meltdown-checker tells us that STIBP is at default settings (conditionally enabled), we can infer that the attack vector we verified with the empirical tools affects the U-U and S-P use cases but it does not affect U-K.

E. Static analysis

A clear limitation of information gathering tools considered in this work is that, for attacks such as Spectre-PHT, their verification is at best a coarse approximation. This is because none of the tools inspects the target application at code level to verify if proper mitigations are inserted (e.g. lfencing sensible branches or branchless masking). While no comprehensive static analysis tool is available, known techniques

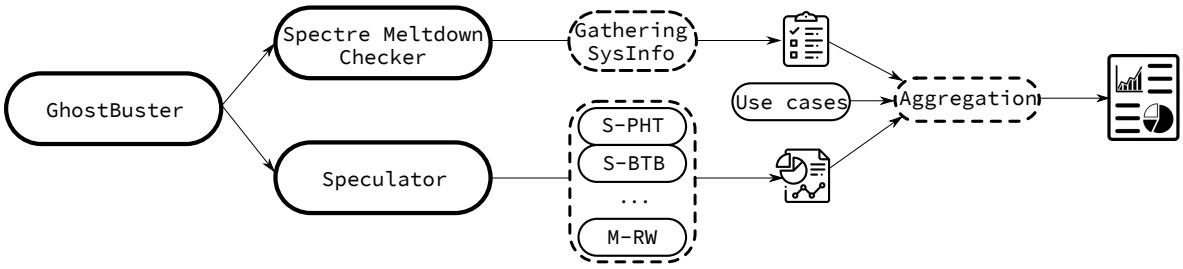

Fig. 1. GhostBuster's overview. GhostBuster leverages spectre-meltdown-checker and our modified version of speculator to assess the system security using both known methodologies, gathering and empirical. Then, it aggregates the results in a final report factoring in also the various use cases we identified to give a more accurate picture to the user. With solid circles we describe the major components of GhostBuster while with dotted circles we highlight operations performed.

such as lfence counting (which is implemented in spectre-meltdown-checker only for the current kernel image) can help determine whether such protections are in place. We recommend future work to integrate this type of static analysis to be able to inform users better, in particular with respect to U-U use cases targeting important user space programs and libraries, such as OpenSSH or OpenSSL.

```
========= SPECTRE BTB cross address-space =========
* Attack success rate (synthetic test): 75.00% (spatial
    colocation), 56.00% (temporal colocation)
* Attack vector: Present
* Difficulty: Low - Practical attacks demonstrated in user
    -to-kernel and user-to-user use cases.
  ------ USE CASES -------
* S-P: This use case does not apply for this attack.
* U-U: Check that the target process is compiled with lfence
     or retpoline.
    IBPB is conditionally enabled: check that the target
        process invokes it with prctl/seccomp.
    STIBP is conditionally enabled: check that the target
        process invokes it with prctl/seccomp.
* U-K: This kernel is not vulnerable: Full retpoline + IBPB
    are mitigating the vulnerability.
* G-G: Enable STIBP and IBRS system-wide on the guest
    machine. If STIBP and IBPB are conditionally enabled,
    check that the target process invokes them with prctl/
    seccomp or is compiled with retpoline.
* G-H: Check that IBRS is enabled on the host, or that it is
     compiled with retpoline and uses IBPB.
* H-SGX: Not vulnerable: IBRS is enabled.
```

Listing 6. GhostBuster sample output for Spectre BTB cross address-space

VI. GhostBuster

Given that none of the four available tools provide a complete and consumable answer as to whether a system is vulnerable to the various classes of transient execution attacks, we prototype a new tool, GhostBuster, shown in Figure 1, that takes into account the recommendations presented in the previous section and provides a system administrator with accurate information to decide whether their system is vulnerable.

The tool is built on the foundation of *empirical* and *information gathering* methodologies combined, as a result of the insights collected during our analysis. GhostBuster is a meta-tool combining a modified version of speculator and spectre-meltdown-checker, which allows us to use the best of each available approach. Another key difference from existing tools is that GhostBuster provides information explicitly based on use cases presented in Section III-A. The use case information is integrated with speculator and spectre-meltdown-checker outputs during the *Aggregation* phase as depicted in Figure 1.

In GhostBuster, the first tool we leverage is an enhanced version of speculator. For GhostBuster, we include two sets of empirical tests, the PMC based we used during our analysis in Section II-C2 and a second set, a cache based series of tests similar to the ones used in transientfail. The two set of tests for empirical verification are necessary to make sure we can have a fallback mechanism when one of the two is not available, once again making the best out of available approaches to avoid the pitfalls we identified. As mentioned in Section IV-A, PMC-based tests cannot be used in a virtualized environment because such interface is not exported to the guest. Similarly, there are cases in which the system has too much LLC activity, making the tests using the cache unreliable and therefore requiring the PMC-based tool. When the system setup allows so, GhostBuster runs both empirical test sets to have confirmation of the results and detects any mismatch. If GhostBuster detects a huge amount of LLC activity, it prompt a warning to the user to signal that possible problems can arise while running transientfail.

GhostBuster uses the spectre-meltdown-checker output in the analysis phase, for its comprehensive report on supported mitigations on the target system, together with their activation status. This enables GhostBuster to connect the synthetic results provided by the tests with the actual use cases. Subsequent to information gathering and analysis, GhostBuster presents the results based on each use case. It presents information about the difficulty level D of the attack, which we provide based on whether real-world attacks using that attack vector exist and how easy it is for the attacker to meet the requirements for the attack. Formally, we compute it as follows:

$$D = 100 - \left[40\ rw + \left(60 - \sum_{j=0}^{N} W_j R_j \right) \right] \tag{1}$$

$$= \begin{cases} [0, 40) & \text{Low} \\ [40, 70) & \text{Medium} \\ [70, 100] & \text{High} \end{cases} \tag{2}$$

where $rw \in \{0, 1\}$ indicates whether the considered attack has a real world instance, $R_j \in \{0, 1\}$ indicates if the identified attack requirement j is present for the attack, N is the total number of possible requirements found for a certain type of attack, and W_j represents the difficulty weight for each of the requirements, that for the sake of simplicity we set $W_j = \frac{60}{N}$ for each requirement j.

When possible, GhostBuster reports the system status,

vulnerable or not vulnerable. In cases when such a conclusion cannot be drawn, as may be the case with Spectre-PHT or Spectre-BTB where the attack and mitigations are program dependent, `GhostBuster` provides a checklist that the user can follow to verify the security of such application. We prefer to provide a checklist for some of the cases instead of trying an automatic approach because there are no error-free methods to detect, for instance, if an application is instrumented with SLH against Spectre-PHT. False positive results might induce a false sense of security which is against the principles `GhostBuster` is designed with, so we suggest the user what exactly requires manual verification instead.

Listing 1 shows the output of `spectre-meltdown-checker` and Listing 5 and Listing 6 show the output of `GhostBuster` in relation to the Spectre-BTB attack. These outputs are taken from the same machine in the same settings. `spectre-meltdown-checker` provides raw information about the mitigations status (e.g. present/not present). For instance, it confirms the availability of mitigations such as IBRS and IBPB, and marks them as active. Also, it shows that the kernel is compiled with retpoline. Finally it informs the user that the system is *not vulnerable* because both mitigations IBRS and IBPB are present on the machine. We deem this output to be misleading because the same machine tested under `speculator` is reported vulnerable to the Spectre-BTB attack vector despite the picture depicted by `spectre-meltdown-checker` output. In practice, this means existing known attacks such as SMoTherSpectre [2] leaking bytes from OpenSSL are feasible on this machine.

In contrast, `GhostBuster` provides more precise information. First, it provides the attack success rate that empirical tests have obtained and it confirms the presence of the attack vector. This information is retrieved thanks to our enhanced version of `speculator` fork.

Second, based on the output of `spectre-meltdown-checker` we are able to provide a more detailed view regarding each one of the use cases. It is possible to notice that `GhostBuster` considers the system protected against Spectre-BTB under the U-K use case. This confirms the output of `spectre-meltdown-checker` from Listing 1 that strictly focuses on the kernel protection from transient execution attacks. Instead, for cases such as U-U, `GhostBuster` recognizes that there are settings (e.g. same address space) in which no information regarding mitigations is available and therefore no final decision can be taken based on available information. In such cases `GhostBuster` provides suggestions such as to verify that the target application is compiled with lfence/retpoline, thereby not misleading the user into a false sense of security. Third, `GhostBuster` considers, when necessary, the various attack settings (cAS/cHT/sAS). For Spectre-BTB, `GhostBuster` provides different outputs for same and cross address space and adjusts the recommendation accordingly to the scenario. For instance, for the U-U use case in the cross address space setting, it suggests to the user to verify if the target application requests to enable STIBP and IBPB to the kernel through either `seccomp` or `prctl` interface. In fact, it even distinguishes between the need for STIBP (mitigating temporal colocation, cHT) and for IBPB (mitigation spatial colocation, cAS) depending on whether none, either, or both of the two empirical tests are successful. In the output shown here, both tests pass and both attack vectors are present, therefore the recommendation is to enable both STIBP and IBPB.

Another example for comparison between `GhostBuster` and `spectre-meltdown-checker` is provided by Listing 3 and Listing 4 for the Spectre-STL attack. Here, `spectre-meltdown-checker` detects that the system supports Speculative Store Bypass (SSB) and simply declares the system *not vulnerable* based on this information. In reality, the system should be considered vulnerable for most of the use cases because the SSB mitigation is enabled only conditionally, which is reflected in the `GhostBuster` output. Therefore, in use cases such as U-U and S-P, the target application must be checked and forced to use either `seccomp` or `prctl` which would enforce the mitigation for the current process. Nevertheless, `GhostBuster` also makes sure to let the administrator know that this is a minor threat, given that no known attacks exist to this date.

This type of output comparison is valid for all the supported attacks, which we do not report for sake of space. As shown, `GhostBuster` enhances current tools output to include use cases and more targeted information, thus standing out as a more accurate and usable tool. Although our current `GhostBuster` implementation is meant for x86/x86_64 Linux machines, the principles incorporated and its design remain valid for other architectures (e.g ARM) and other Operating Systems (e.g. Windows).

To conclude, `GhostBuster` provides a more detailed and accurate view of a system's vulnerability to transient execution attacks by simply combining the best features of existing approaches and presenting them in an understandable way. While for certain use cases the assessment is binary, vulnerable or not vulnerable, for others, `GhostBuster` guides the user on testing whether the target application meets the correct safety requirements against an attack under the considered use case.

VII. Conclusion

In this work, we look into current tools and methodologies that aim to help system administrators and users to verify the exposure of their systems to transient execution attacks. We test and gather results for each of the tools on 17 different platforms including both AMD and Intel CPUs. We find that current techniques do not cover important settings of transient execution attacks and often provide misleading outputs. While empirical tools focus solely on verifying the presence of the attack vector on the system in a synthetic manner, the information gathering tools check for the presence of mitigations and kernel information that hint the protection of the system but come short in clearly specifying the use cases in which the system is actually protected. We report our results in relation with 6 different use cases and underline the major pitfalls found in each. Based on our experience, we propose 5 main recommendations and propose a meta-tool, `GhostBuster`, which incorporates our recommendations. `GhostBuster` combines the best of each tool to enhance the accuracy and clarity of the results, guiding system administrators and users towards the right actions to properly protect their system without unnecessarily sacrificing performance.

ACKNOWLEDGEMENT

This work was partially-supported by National Science Foundation under grant CNS-1703454, and ONR under the "In Situ Malware" project.

REFERENCES

[1] ARM LIMITED, "Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism," 2018.

[2] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019.*, 2019, pp. 785–800. [Online]. Available: https://doi.org/10.1145/3319535.3363194

[3] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019, extended classification tree at https://transient.fail/.

[4] C. Carruth, "Speculative Load Hardening," https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html, 2018.

[5] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019, pp. 142–157.

[6] J. Corbet, "Meltdown/spectre mitigation for 4.15 and beyond," https://lwn.net/Articles/744287/, 2018.

[7] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," 09 2018.

[8] Gartner, "Gartner says worldwide iaas public cloud services market grew 31.3in 2018," 2018. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2019-07-29-gartner-says-worldwide-iaas-public-cloud-services-market-grew-31point3-percent-in-2018

[9] Google, "Google safeside project," 2019. [Online]. Available: https://github.com/google/safeside

[10] J. Horn, "Speculative Execution, variant 4: speculative store bypass," 2018.

[11] Intel, "Deep dive: Indirect branch predictor barrier," https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-predictor-barrier, 2018.

[12] ——, "Deep dive: Indirect branch restricted speculation," https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-restricted-speculation, 2018.

[13] ——, "Deep dive: Single thread indirect branch predictors," https://software.intel.com/security-software-guidance/insights/deep-dive-single-thread-indirect-branch-predictors, 2018.

[14] ——, "Intel Analysis of Speculative Execution Side Channels, Revision 4.0," July 2018.

[15] ——, "Q2 2018 Speculative Execution Side Channel Update," May 2018.

[16] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," https://people.csail.mit.edu/vlk/spectre11.pdf, 2018.

[17] P. Kocher, 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 *IEEE Symposium on Security and Privacy*, 2018.

[18] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *USENIX Workshop On Offensive Technologies*, 2018.

[19] S. Lesimple, "spectre-meltdown-checker script," 2018. [Online]. Available: https://github.com/speed47/spectre-meltdown-checker

[20] 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 *USENIX Security Symposium*, 2018.

[21] 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: ACM, 2018, pp. 2109–2122. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243761

[22] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: A tool to analyze speculative execution attacks and mitigations," in *To appear in proceedings of the 35th Annual Computer Applications Conference ACSAC*. San Juan, PR, USA: ACS Association, Dec. 2019. [Online]. Available: https://www.openconf.org/acsac2019/modules/request.php?module=oc_program&action=summary.php&id=223

[23] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus, "Two methods for exploiting speculative control flow hijacks," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/mambretti

[24] 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.

[25] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," https://support.google.com/faqs/answer/7625886, 2018.

[26] J. Van 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 *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018, see also technical report Foreshadow-NG [30].

[27] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[28] 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, intel Bounty Reward (Highest To Date) , Pwnie Award Nomination for Most Innovative Research, CSAW Best Paper Award Runner-up. [Online]. Available: https://mdsattacks.com

[29] VUSec, "mdstool-cli tool," 2019. [Online]. Available: https://github.com/vusec/ridl

[30] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," *Technical report*, 2018, see also USENIX Security paper Foreshadow [26].

[31] Y. Xiao, Y. Zhang, and R. Teodorescu, "SPEECHMINER: A framework for investigating and measuring speculative execution vulnerabilities," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/speechminer-a-framework-for-investigating-and-measuring-speculative-execution-vulnerabilities/

[32] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," IEEE, pp. 79–93, 2009.