Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel

(*Extended Version*)

Brian Johannesmeyer*‡, Jakob Koschel*‡, Kaveh Razavi†, Herbert Bos*, and Cristiano Giuffrida*

*VU Amsterdam †ETH Zurich

‡Equal contribution joint first authors

***Abstract*—Due to the high cost of serializing instructions to mitigate Spectre-like attacks on mispredicted conditional branches (Spectre-PHT), developers of critical software such as the Linux kernel selectively apply such mitigations with annotations to code paths they assume to be dangerous under speculative execution. The approach leads to incomplete protection as it applies mitigations only to easy-to-spot gadgets. Still, until now, this was sufficient, because existing gadget scanners (and kernel developers) are pattern-driven: they look for known exploit signatures and cannot detect more generic gadgets.**

In this paper, we abandon pattern scanning for an approach that models the essential *steps* used in speculative execution attacks, allowing us to find more generic gadgets—well beyond the reach of existing scanners. In particular, we present Kasper, a speculative execution gadget scanner that uses taint analysis policies to model an attacker capable of exploiting arbitrary software/hardware vulnerabilities on a transient path to control data (e.g., through memory massaging or LVI), access secrets (e.g., through out-of-bounds or use-after-free accesses), and leak these secrets (e.g., through cache-based, MDS-based, or port contention-based covert channels).

Finally, where existing solutions target user programs, Kasper finds gadgets in the kernel, a higher-value attack target, but also more complicated to analyze. Even though the kernel is heavily hardened against transient execution attacks, Kasper finds 1379 gadgets that are not yet mitigated. We confirm our findings by demonstrating an end-to-end proof-of-concept exploit for one of the gadgets found by Kasper.

I. Introduction

Ever since Meltdown and Spectre burst onto the scene in January 2018 [23, 32, 36], transient execution vulnerabilities have had the security community scrambling for solutions. While some of the vulnerabilities, such as Meltdown, can be mitigated fairly easily [16], this is not the case for others. In particular, Spectre-PHT (commonly referred to as Spectre-V1)—which leaks secrets by abusing the transient execution that follows a mispredicted conditional branch—cannot be fully eradicated without crippling performance. One possible mitigation is to forbear all speculation after a conditional branch. However, since speculative execution in modern CPUs is essential to performance and conditional branches are everywhere, it is crucial that such an expensive mitigation be applied only where necessary. The attack itself requires specific Spectre-PHT *gadgets*: that is, vulnerable branches which can indeed leak secrets through the microarchitectural state. Therefore, to maintain acceptable performance, we should only serialize or otherwise instrument these gadgets.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Listing 1: Spectre-PHT BCB pattern, where attacker data bypasses an array bounds check in order to leak secret data.

Beyond pattern-matching. However, current methods to identify gadgets are limited as they are entirely pattern-driven. By searching for specific features of well-known Bounds Check Bypass (BCB) patterns (Listing 1)—e.g., suspicious copies from userspace [11], potential out-of-bounds accesses [43], or attacker-dependent memory accesses [47]—they only approximate the presence of BCB gadgets. Our experiments show that today's state-of-the-art scanners [43, 47] yield a false positive rate of 99% (Section IX-A). In other words, 99% of the code snippets identified as gadgets cannot actually lead to information disclosure. Adding expensive defenses to such code snippets incurs substantial and unnecessary overhead. More, the false negative rate is up to 33%. In other words, they miss many vulnerable branches that should be protected.

Furthermore, existing approaches are limited in scope. They all broadly assume the same primitives as the traditional BCB pattern: direct attacker input, an out-of-bounds secret access, and a cache-based covert channel. Such primitives are of little use in the hardened Linux kernel since the large majority of BCB gadgets have been mitigated. However, this does not mean the kernel is free of Spectre-PHT gadgets. Far from it: the problem goes much deeper than BCB patterns. In reality, attackers do not care about patterns; they just want to find *any* instructions in the wake of *any* conditional branch, controllable by *any* means, which access secrets in *arbitrary* ways, and leak the secrets through *any* covert channel.

This version includes the extended versions of Appendices C and E, which were included in the original version submitted and accepted at NDSS 2022.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24221
www.ndss-symposium.org

Step 1: *Inject* controlled values into transient execution.
Step 2: Force the execution to *access* a secret.
Step 3: Force the execution to *leak* the secret.

Fig. 1: Essential steps in a Spectre attack.

A principled approach. In this paper, we propose a novel approach for finding vulnerable gadgets in the kernel, abstracting away all pattern-specific details and instead precisely modeling the *essential steps* of a Spectre-PHT attack shown in Figure 1. Around these steps, we use targeted taint analysis policies to generically model the effects of *arbitrary hardware/software vulnerabilities* on a transient path.

To evaluate the approach, we present KASPER, a Spectre-PHT gadget scanner. By modeling the effects of vulnerabilities on a transient path—e.g., memory errors [13, 32], load value injection [55], cache-based covert channels [32], MDS-based covert channels [10, 51, 56], and port contention-based covert channels [9, 19]—KASPER finds gadgets well beyond the scope of existing approaches.

Moreover, rather than focusing on user processes (to which our approach is easily applicable), we developed our techniques specifically for the Linux kernel—a high-value target like few other programs; since the kernel has access to all memory in the system, a kernel attacker can target data from any of the running processes in the system. Existing kernel gadget scanners however, are all based on static analysis [11, 30], which previous work observes is imprecise [43, 47]. Instead, we take a dynamic analysis-based approach, which simply requires us to build the kernel with KASPER support, fuzz the syscall interface (to simulate the possible coverage from a user-to-kernel attacker), then KASPER will report gadgets at runtime. Even though the kernel has undergone intensive scrutiny and mitigation efforts to neuter all gadgets, we still find 1379 gadgets in an automated fashion—including many gadgets which would be non-trivial to find statically.

Furthermore, we present a case study of a gadget found by KASPER which is pervasive throughout the codebase and non-trivial to mitigate. In total, we have found 218 instances of the demonstrated gadget sprinkled all over the kernel. We demonstrate the efficacy of our approach by presenting a proof-of-concept exploit of this gadget.

Contributions. We make the following contributions:

- We present taint-assisted generic gadget scanning, a new approach to identify pattern-agnostic transient execution gadgets that stem from arbitrary software/hardware vulnerabilities on a transient execution path.
- We present KASPER, an implementation[1] of our approach for the Linux kernel.
- We evaluate KASPER on a recent version of the Linux kernel to identify 1379 previously unknown transient execution gadgets and present a proof-of-concept exploit for one of the gadgets. The Linux kernel developers are currently working on mitigations for the disclosed gadgets and requested access to KASPER for mainline kernel regression testing moving forward.

TABLE I: Overview of the possible primitives of a Spectre gadget. KASPER models the primitives in bold.

Speculation	Attacker input	Secret output
PHT	ARCH:[a] **USER**, FILE, NET, DEV, **MASSAGE**	**CACHE**
BTB		**MDS**
RSB		**PORT**
STL	**LVI**	AVX
	FPVI	

[a] We only list the set of ARCH inputs that are relevant to the kernel.

The rest of this paper is organized as follows. Section II presents background on the attack primitives modeled by KASPER. Section III describes the threat model. Section IV defines the problem scope for KASPER. Section V describes the design of KASPER at a high level. Section VI explains speculative emulation, including unique implementation challenges posed by the kernel. Section VII explains KASPER's vulnerability detectors and the taint policies which model their effects. Section VIII briefly describes implementation details. Section IX evaluates the efficacy of KASPER compared to existing approaches and KASPER's gadgets found in the kernel. Section X discusses the limitations of our approach. Section XI presents a case study of a gadget found by KASPER. Finally, Section XIII concludes.

II. Background

In this section, we will first provide a background on the components involved in a Spectre attack and the defenses which combat them, motivating the need for Spectre-PHT gadget scanners. Then, we will provide background on previous gadget-scanning tools, highlighting the need for a pattern-agnostic gadget scanner.

A. Speculative execution attacks and defenses

Spectre attacks exploit the fact that modern processors predict the outcome of operations such as conditional branches, and speculatively continue executing as if its prediction is correct. If it turns out the prediction was incorrect, the processor reverts the results of any speculative operations and restarts from the correct state. The modifications made by the incorrect path to the microarchitectural state, however, can be examined by an attacker using a covert channel to leak sensitive information.

We propose that the underlying primitives used by a Spectre gadget—i.e., the type of speculation it abuses, the type of attacker data it depends on, and the type of leakage it exploits—define a Spectre variant. The interactions between the different primitives affect whether the variant is indeed exploitable and whether it is easily mitigated. We summarize these primitives in Table I.

Speculation type. Spectre variants based on speculation from anything other than the *Pattern History Table*—that is, Spectre-*BTB* [32], Spectre-*RSB* [34, 40], and Spectre-*STL* [24]—are easily mitigated with relatively low overhead using microcode updates [26] or software updates such as retpolines [25] and static calls [64]. Unfortunately, this is not so for Spectre-PHT.

[1] KASPER is available at https://www.vusec.net/projects/kasper [2].

Spectre-PHT gadgets can be mitigated by adding an `lfence` instruction after conditional branches; this approach does not scale, however, as adding an `lfence` after every conditional branch would incur up to a massive 440% overhead [42]. Hence, rather than avoiding speculation altogether via `lfence`, Spectre-PHT is better addressed by mitigating specific speculative operations—that is, either at the point where attacker data is used or where secret data is leaked.

Attacker input type. Attacker data may reach a kernel gadget in a variety of ways—either via architectural or microarchitectural means. First, an attacker may pass data to the kernel via well-defined interfaces such as *userspace*, *files*, the *network*, or malicious *devices*. Second, data can come from such places as normal, but then an attacker may inject it into a victim kernel thread via targeted *memory massaging* [13, 14, 61, 65]—wherein the attacker lands the data into a specific place on the kernel's stack or heap, in the hopes that later on, a bug such as an out-of-bounds read or an uninitialized read (bugs that are relatively common on a transient path) will eventually use the malicious data. Third, using thread that shares a simultaneous multithreaded (SMT) core with a kernel thread (i.e., where the core executes instructions from both the attacker thread and the victim thread at the same time), the attacker may inject data into the victim's transient path via *load value injection (LVI)* [55]—wherein the attacker issues a sequence of faulting stores, filling the CPU's load port with unresolved data dependencies; meanwhile, if the kernel simultaneously loads from the same faulting address, the CPU will inadvertently serve malicious data to the kernel. Concurrent to our work, *floating point value injection (FPVI)* [48] similarly demonstrates this issue for floating point values. Note that when we will discuss gadget exploitability, we will group together variants using *architectural* input, since a Spectre gadget is agnostic to architectural-level semantics, and will execute the same regardless of whether the input comes from e.g., userspace or memory massaging.

A widely-used mitigation to pacify attacker input is to prevent certain transient array accesses from accessing secret data out-of-bounds by forcing the access to stay in-bounds via a masking operation. The Linux kernel uses user pointer sanitization and the macro `array_index_nospec` for this purpose. This approach however, does not generalize well to non-array transient accesses.

Secret output type. Secrets may leak from a kernel gadget in a variety of ways. First, a gadget may rely on a *cache*-based channel [32], wherein the victim dereferences a secret, thereby leaving a trace on the data cache (or TLB); an attacker can then recover this secret through methods such as FLUSH+RELOAD [62]. Second, a gadget may rely on a *microarchitectural data sampling (MDS)*-based channel [10, 51, 56], wherein the victim simply accesses a secret, causing the CPU to copy the secret into its load buffer (or line fill buffer, store buffer, etc.); meanwhile, an attacker can co-locate a thread on a SMT core and issue a conflicting load. As a result, the CPU will inadvertently serve the secret data to the attacker, who can then use it to leave a trace on their own FLUSH+RELOAD buffer for recovery. Third, a gadget may rely on a *port contention*-based channel, wherein the victim—depending on a secret—either executes one set of instructions or another; meanwhile, an attacker can co-locate a thread on a SMT core and issue instructions that compete for the same execution units (i.e., ports) as the victim's instructions. Then, the attacker can use timing information to infer which instructions the victim executed, and hence, learn a bit of the secret [9, 19]. Finally, a gadget may rely on an *AVX*-based channel, which exploits the timing of AVX2 instructions [52].

The kernel flushes CPU buffers to mitigate same-thread MDS channels. Hardware updates for the most recent generation of CPUs mitigate cross-thread MDS (and LVI) channels. For the same protections, older CPUs must disable hyperthreading, resulting in massive performance penalties (not the default on Linux). The reality that these defenses are only on by default on the newest CPUs, and that PHT gadgets cannot be systemically mitigated via `lfence`, `array_index_nospec`, and user pointer sanitization, highlights the pressing need for Spectre-PHT gadget scanners.

B. Gadget scanning

Existing gadgets scanners, however, cannot accurately identify gadgets that stray even slightly from the basic BCB pattern in Listing 1.

Static analyses. Existing static analyses find gadgets through pattern-matching of source code [11, 21], pattern-matching of binaries [15], static taint analysis [59], and symbolic execution [22, 58]. Concurrent work [30]—which goes beyond the BCB pattern, and instead targets a type-confusion pattern—similarly uses such approaches. These approaches, however, all suffer from the fundamental limitation of static analysis: assumptions to compensate for unknowns at compile time will severely hamper soundness and/or completeness. For example, consider the gadget in Listing 2a, which loads its input from a source that is unknown at compile time. In theory, an advanced analysis such as symbolic execution could deduce whether `*ptr` is indeed attacker-controllable by verifying the controllability of *every possible* value in *every possible* location of `*ptr`; however, this kind of analysis cannot scale to the complex and massive kernel codebase [65], so it instead must ultimately either assume that `*ptr` is attacker-controllable (and risk false positives) or that it is not (and risk false negatives). Such scalability issues are inherent to any sufficiently complex static analysis, leading to imprecise points-to analyses, inaccurate call graph extractions, and ultimately, imprecise gadget identifications. In Appendix D, we describe a gadget found by our dynamic analysis which relies on an indirect call, thereby thwarting static call graph extraction, and hence, identification by static analysis altogether.

Dynamic analyses. To overcome the limitations of static analysis, recent work instead opts for dynamic analysis. Their approaches however, fall short since they are pattern-driven and do not model the underlying semantics of gadgets.

For example, since the BCB pattern has an out-of-bounds access, SpecFuzz [43] uses code sanitizers to report as gadgets any speculative out-of-bounds accesses. By targeting this single behavior however, it incurs false negatives for gadgets such as the one in Listing 2b, whose *leak* instruction is in-bounds.

```
x = *ptr;
if (x < size) {
  y = arr1[x];
  z = arr2[y]; }
```

(a) Gadget that is difficult to detect statically because it is unclear whether `*ptr` is attacker-controllable.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y & MASK]; }
```

(b) Gadget that is undetectable by SpecFuzz [43] because its in-bound *leak* eludes code sanitizers.

```
x = 1000;
if (x < size) {
  y = arr1[x];
  z = arr2[y]; }
```

(c) Gadget that is not controllable by an attacker, yet falsely reported by SpecFuzz [43] because it yields an out-of-bounds access.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x & MASK];
  z = arr2[y]; }
```

(d) Gadget that is mitigated by the masking operation, yet falsely reported by SpecTaint [47] because there is a direct dataflow from `x` to `y` to `arr2[y]`.

```
if (addr_is_mapped(ptr)) {
  x = *ptr;
  y = arr1[x];
  z = arr2[y]; }
```

(e) Gadget that existing approaches cannot detect. `*ptr` gives a transient page fault, so an attacker can *inject* data for `x` via LVI [55].

Listing 2: Gadgets that differ slightly from the basic BCB pattern in Listing 1, and therefore foil existing approaches.

Furthermore, it incurs false positives for any unrelated out-of-bounds accesses, such as the one in Listing 2c, even though it is entirely uncontrollable by an attacker.

Another property about the BCB pattern is that there is a direct dataflow from an attacker-controllable value, to a secret, and finally to a *leak* instruction. Targeting this single property, SpecTaint [47] taints attacker-controllable values (`x`), then taints as a secret any attacker-dependent loads (`y`), then reports as a gadget any secret-dependent accesses (`arr2[y]`). This policy however, assumes very specific patterns and also incurs false positives since not every attacker-dependent load accesses secret data. For example, consider the attacker-dependent load in Listing 2d: even though the masking operation prevents it from accessing secret data, SpecTaint falsely reports this mitigated gadget as an exploitable gadget.

III. Threat Model

We consider a local unprivileged attacker with the ability to issue arbitrary system calls to a target kernel free of (exploitable) software bugs. The attacker aims to gain a memory leak primitive by exploiting a transient execution gadget in the kernel. As an operating system kernel, we focus on a recent Linux kernel (in our case 5.12-rc2) with default configurations, including all the mitigations against transient execution attacks enabled, such as user pointer sanitization, `lfences`, `array_index_nospec`, retpolines, static calls, etc. Additionally—although overlooked by the Linux kernel's transient execution threat model [7, 8], which only considers attacker input from userspace and MDS/cache-based covert channels—we consider an attacker able to: (1) inject data via memory massaging, (2) inject data via LVI, and (3) exfiltrate data via port contention-based covert channels. Note that on the most recent generation of CPUs, LVI and MDS are mitigated in-silicon to a large extent, so our results for LVI and MDS do not apply to the full extent to such CPUs.

IV. Problem Analysis

Not only do existing approaches fail to identify gadgets which stray slightly from the basic BCB pattern (Section II-B), they do not even attempt to model primitives beyond those it uses—i.e., where an attacker directly passes data to the victim, causing a cache-based leak. For example, consider the gadget in Listing 2e, where an attacker can inject data via LVI [55]. No existing approach attempts to model LVI, and hence, gadgets such as these are missed. In reality, attackers have many such primitives at their disposal (Table I)—such as memory massaging, LVI, MDS, and port contention—all of which may yield gadgets that fall outside the scope of existing approaches.

TABLE II: Exploitability of the Spectre variants that are composed of the primitives which Kasper models. Despite the signal on 4 variants, existing scanning techniques target only PHT-Arch-Cache.

Spectre variant	Described in:	Verified signal?	Existing scanning techniques?
PHT-Arch-Cache	[32]	✓	✓
PHT-Arch-MDS	[10]	✓	-
PHT-Arch-Port	[19]	✓	-
PHT-LVI-Cache	This paper[a]	✓	-
PHT-LVI-MDS	None[a]	-[b]	-
PHT-LVI-Port	None[a]	-	-

[a] The demonstrated LVI attack [55] exploits an LVI-Cache gadget.
[b] We verified a signal for LVI-MDS, but not for PHT-LVI-MDS.

To reason about the various combinations of primitives which are possible in a Spectre gadget, we express Spectre variants as a triple in terms of these primitives. For example, we consider the original Spectre-PHT attack [32] which exploited the BCB pattern to be a PHT-Arch-Cache gadget because it: (1) exploits prediction from the PHT, (2) depends on architecturally-defined attacker input, and (3) leaks the secret via the cache. Similarly, Fallout [10] uses a PHT-Arch-MDS gadget and SpectreRewind [19] uses a PHT-Arch-Port gadget. Other Spectre attacks, including those using other speculation types, can be expressed as a triple in this way; e.g., SMoTherSpectre [9] exploits a BTB-Arch-Port gadget. Furthermore, even non-Spectre attacks can be expressed as a tuple of the last two primitives; e.g., a standard cache attack [45] exploits an Arch-Cache gadget.

Finally, assuming a gadget scanner can model all such primitives, we would first need to verify that all the possible variant combinations are indeed exploitable. In Table II, we summarize the exploitability of the Spectre variants made up of the primitives modeled by Kasper. The first three rows, as described above, are based on attacks from previous work. Beyond existing work, we were able to verify a signal for a PHT-LVI-Cache gadget, but not for a PHT-LVI-MDS gadget or a PHT-LVI-Port gadget. Hence, Kasper models the first four variants of the table.

V. Overview

The key insight to our approach is that by generically modeling the vulnerabilities that an attacker can use in each step of a Spectre attack, we can precisely identify gadgets. In particular, we use: (1) *speculative emulation* to model transient execution, (2) *vulnerability detectors* to model various software and hardware vulnerabilities, (3) *dynamic taint analysis* to model the architectural and microarchitectural effects of such vulnerabilities, and (4) *fuzzing* to model the possible coverage of an attacker. Figure 2 presents an example of how these components interact to identify a gadget in a system call handler.

Modeling transient execution. To model branch mispredictions, we invert conditional branches at runtime and emulate the corresponding transient execution by taking a checkpoint at the point of 'misprediction' and executing the code that would be executed speculatively. We roll back to resume normal execution when the speculative window closes.

Speculative emulation is not trivial in general and this is especially true for speculative emulation in the kernel, where complexities such as device interactions, exceptions, and inline assembly all pose challenges. We explain how our approach overcomes these challenges in Section VI.

Modeling software and hardware vulnerabilities. By modeling transient execution at runtime, we expose transient code paths to runtime checkers to generically identify software and hardware vulnerabilities which would otherwise remain undetectable.

Our current prototype uses: (1) a memory error detector to target *software vulnerabilities* in the shape of transient out-of-bounds and use-after-free accesses, (2) an LVI detector to target *hardware vulnerabilities* via transient faulting accesses, and (3) a covert channel detector for *hardware vulnerabilities* via cache-based, MDS-based, or port contention-based channels. We explain our detectors in more detail in Section VII-A.

Modeling the effects of transient vulnerabilities. By detecting software and hardware vulnerabilities, we can design taint policies around the essential steps of a Spectre attack (Listing 1) to reason about the effects of such issues.

For example, our policies may handle a transient invalid load in different ways: (1) our LVI detector may taint the load with an `attacker` label to indicate that the attacker may *inject* the value via LVI; (2) our memory error detector may taint the load with a `secret` label to indicate that it may *access* arbitrary memory; or (3) our covert channel detector may report the load as an MDS-LP [56] covert channel to indicate that it may *leak* `secret` data. We explain how our taint policies reason about the interactions between different vulnerabilities on a transient path in Sections VII-B–VII-D, including a summary of the policies in Figure 3.

```
  x = get_user(ptr);
  // Sets in_checkpoint
  if (!in_checkpoint) new_checkpoint();
L1:
  if ((x < size)
      XOR in_checkpoint) {
    y = arr1[x];
    z = arr2[y];
  }
  // Unsets in_checkpoint
  if (in_checkpoint) {rollback(); goto L1;}
```

Listing 3: Conditional branch instrumented to enable speculative emulation.

Modeling the possible coverage of an attacker. By modeling the effects of transient vulnerabilities in the kernel, we can use a user-to-kernel fuzzer to only model the vulnerabilities that are reachable by a userspace attacker issuing arbitrary syscalls. We describe the fuzzer and implementations details in Section VIII, including a summary of the end-to-end pipeline in Figure 4.

VI. Speculative Emulation

To emulate speculative execution, we need to be able to execute possible execution paths that would otherwise not be executed architecturally. Specifically, on a branch misprediction, the processor first executes the *wrong* code path until the speculation is eventually squashed. The processor then continues executing the correct side of the branch. To model such branch mispredictions, we invert conditional branches at runtime in software and to be able to squash the incorrect execution path, we rely on software-based memory checkpointing. Thus, at runtime we will first execute the wrong code path to emulate speculation and when speculation is squashed, we execute the correct code path.

Listing 3 shows how we instrument the example of Listing 1 to support speculative emulation. If `x` $\geq$ `size` at runtime, we:

1) Start a checkpoint immediately before the branch via the call to `new_checkpoint`.
2) Simulate a branch misprediction by flipping the branch via `XOR in_checkpoint`.
3) Emulate speculative execution along the *taken* code path.
4) Simulate a speculative squash by rolling back to the saved checkpoint via `rollback(); goto L1;`
5) Finally, continue normal execution along the correct *not-taken* code path.

A. Transactions and rollbacks

To ensure correct execution, speculative execution is inherently transactional from an architectural point of view; either all the instructions after a predicted branch commit (i.e., correct prediction) or none does. To provide this semantic, we opted for a compiler-based memory checkpoint-restore mechanism to build a notion of transactional execution.

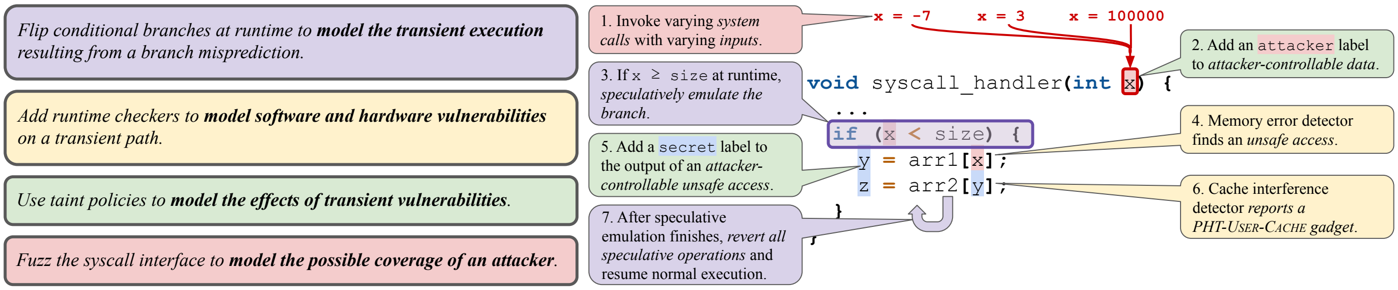

Flip conditional branches at runtime to model the transient execution resulting from a branch misprediction.
Add runtime checkers to model software and hardware vulnerabilities on a transient path.
Use taint policies to model the effects of transient vulnerabilities.
Fuzz the syscall interface to model the possible coverage of an attacker.
1. Invoke varying system calls with varying inputs.
x = -7
x = 3
x = 100000
2. Add an attacker label to attacker-controllable data.
3. If x ≥ size at runtime, speculatively emulate the branch.
void syscall_handler(int x) {
...
if (x < size) {
y = arr1[x];
z = arr2[y];
}
}
4. Memory error detector finds an unsafe access.
5. Add a secret label to the output of an attacker-controllable unsafe access.
6. Cache interference detector reports a PHT-USER-CACHE gadget.
7. After speculative emulation finishes, revert all speculative operations and resume normal execution.

Fig. 2: Components used by our approach and how they interact to detect a PHT-USER-CACHE gadget in a system call handler.

```
x = get_user(ptr);
if (x < size) {
  foo = *bar; // Page fault if bar is invalid
  y = arr1[x]; // Would not execute
  z = arr2[y]; // Would not execute
}
```

Listing 4: Executing past a potential page fault in speculative emulation.

```
static void arch_atomic64_inc(atomic64_t *v) {
  kasper_track_store((void*)&v->counter);
  asm volatile(LOCK_PREFIX "incq %0"
    : "=m" (v->counter)
    : "m" (v->counter) : "memory");
}
```

Listing 5: Enabling a common assembly sequence in speculative emulation.

We implemented our solution with a combination of LLVM compiler passes and a runtime component. On a rollback, we need to restore the original state before the misprediction. Kasper does this by saving all relevant registers when starting a checkpoint and tracking all memory changes in an undo log in preparation for a replay on rollback, similar to prior work [43]. However, doing so in the kernel presents unique challenges which we address in Section VI-B.

Stopping Speculative Emulation. One question we need to answer is when to roll back speculative emulation. Speculative execution is limited to a certain number of *micro-ops* executed depending on the ReOrder Buffer (ROB). At compile time we approximate this behavior with the number of executed LLVM instructions. At the start of every basic block, the number of executed LLVM instructions from the beginning of the checkpoint is compared against a configurable threshold to decide when to abort speculative emulation. While this does not map to the exact behavior of hardware, a reasonable approximation is good enough, and we can easily configure the threshold to be conservative to avoid false positives or more permissive to decrease the likelihood of false negatives. Similarly, we define an upper limit of call depth, simulating the behavior of the Return Stack Buffer (RSB).

Exception handling. Exceptions do not necessarily stop speculative execution, as instructions past an exception can still be executed out-of-order [36, 56]. Our initial evaluation showed that the majority of exceptions raised during kernel speculative emulation are page faults, due to a corrupted memory address within speculation. We hence designed a *page fault suppression* for speculative emulation to execute past raised page faults—and simply stop emulation in the exception handler in the other cases. To avoid page faults, Kasper validates the pointer before dereferencing and replaces it with a valid dummy pointer if it is invalid. Listing 4 shows an example gadget that we would miss without this improvement. Another advantage of dedicated page fault handling is the ability to model common hardware vulnerabilities, as discussed later.

B. Challenges unique to the kernel

Implementing speculative emulation for the kernel introduces new challenges compared to userland. For example, a user program only operates on its own accessible memory while the kernel is able to access the entire range of memory. The kernel is therefore responsible for ensuring full memory integrity while user processes are only aware of their own address space, prompting special treatment, as discussed next. As shown in Appendix A-B, these strategies only contribute to a negligible amount of rollbacks.

Non-conventional memory accesses. The kernel uses device or I/O mapped memory to communicate with external devices. Memory writes to such memory cannot be rolled back by simply writing back the original value. Rollback after writes to such memory ranges would require also rolling back the dedicated device. Since I/O communication does not happen often in most syscall handlers, we gracefully stop emulation in those cases without a big loss in speculative code coverage.

Low-level code and mitigations. The Linux kernel codebase is sprinkled with low-level assembly and transient mitigation code. To ensure speculative emulation correctness, it is important to handle such snippets properly. To this end, Kasper conservatively treats assembly code as a speculative emulation barrier (i.e., forcing rollback) by default and implements dedicated handling for the common assembly snippets to allow their use in speculative emulation. This is to avoid unnecessary rollbacks and maximize speculative code coverage. Listing 5 illustrates an example, with custom handling for atomic increment instructions. Since the assembly is not instrumented, we manually preserve the changed memory location. As for mitigations, we observe the relevant ones (`lfence`, `stac`, etc.)

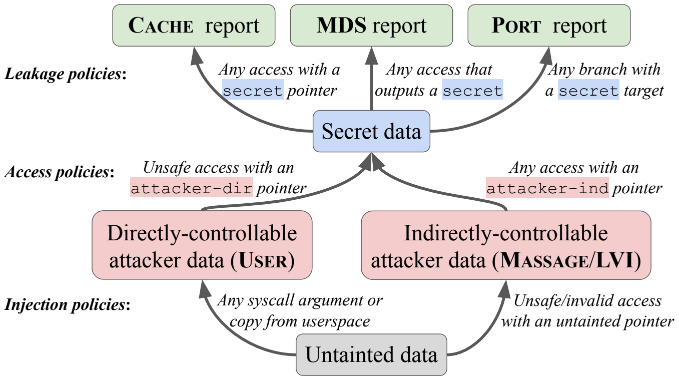

CACHE report
MDS report
PORT report
Leakage policies:
Any access with a secret pointer
Any access that outputs a secret
Any branch with a secret target
Secret data
Access policies:
Unsafe access with an attacker-dir pointer
Any access with an attacker-ind pointer
Directly-controllable attacker data (USER)
Indirectly-controllable attacker data (MASSAGE/LVI)
Injection policies:
Any syscall argument or copy from userspace
Unsafe/invalid access with an untainted pointer
Untainted data

Fig. 3: *Injection*, *access*, and *leakage* taint policies which describe how data rises from an untainted label, to an `attacker` label, to a `secret` label, and finally to a gadget report.

are all implemented in assembly snippets, thus our default assembly handler (i.e., stop speculative emulation) already models them correctly with no special treatment needed.

Concurrency. Unlike many common user programs, the Linux kernel is a highly concurrent piece of software, with multiple threads running in parallel on different CPU cores and hardware (e.g., timer) interrupts causing asynchronous event handling. Taking correct checkpoints in face of concurrency poses a fundamental challenge for the correctness of checkpointing [53] (and speculative emulation in our case). To address this challenge, we disable SMP support in the kernel and force single-core execution. To handle hardware interrupts, we instrument the interrupt handler to: (i) record interrupt information; (ii) stop speculative emulation and resume execution at the last checkpoint; (iii) replay the interrupt as though it was delivered before entering speculative emulation. These steps are all crucial to ensure correct checkpointing *and* kernel execution (i.e., no interrupts are lost).

VII. Taint Policies

To model each of the three steps in Figure 1, we can draw upon previous work that uses information flow policies to detect sensitive data leakage [17, 18, 41, 49, 63], and use a basic framework consisting of three types of policies:

1) Taint any data *injected* by an attacker with an `attacker` label.
2) Taint secret data (i.e., data *accessed* via an `attacker` pointer) with a `secret` label.
3) Report gadgets that *leak* such `secret` values.

However, to detect leakage via generic *transient* execution gadgets, we must refine our framework with taint policies that account for arbitrary vulnerabilities exploited on a transient path that enrich the attacker's capabilities. In particular, with detectors capturing the effects of vulnerabilities in software (i.e., memory errors in the current implementation) or hardware (i.e., LVI, MDS, port contention, and cache covert channels in the current implementation), we are able to model increasingly complex gadgets. In the following, we first detail our current detectors and then discuss how our *taint manager* enforces the detection-based taint policies (for *injection*, *access*, and *leakage*). We summarize our policies in Figure 3.

A. Vulnerability detectors

Memory error detector. Our current memory error detector supports the detection of *unsafe* (i.e., out-of-bounds or use-after-free) accesses on a transient execution path. This is done by running sanitizers during speculative emulation and reporting information about unsafe load/store operations and addresses to the taint manager. Information about unsafe accesses is useful to model a degree of attacker's control on a given address. For instance, if an out-of-bounds detection is taken as evidence that an attacker can make a load address go out-of-bounds at will, that address is a *controlled pointer* that could be used for *injection* (via memory massaging), *access* (via unauthorized reads), and *leakage* (via vulnerabilities such as MDS).

LVI detector. Our current LVI [55] detector supports the detection of *invalid* loads able to trigger LVI on a transient execution path. We tested the (known) LVI triggering conditions and could reproduce only two cases of such invalid loads on a mispredicted branch, also observed in prior work [10]: (i) loads incurring an SMAP fault (i.e., loading a user address with SMAP on—default); (ii) loads incurring a noncanonical address fault (i.e., loading an address outside the canonical 48-bit address space). We also verified these loads can be poisoned by attackers for *injection* of transient values via MSBDS exploitation [10]. To identify such *invalid* loads, we target loads with user/noncanonical addresses (a subset of unsafe loads). However, by default we omit NULL pointer dereferences. Allowing them leads to plenty of usable gadgets, however those require an attacker to map the 0x0 page (i.e., to cause an SMAP fault). The latter is forbidden by default on Linux, but possible on systems with `mmap_min_addr=0`.

Cache interference detector. Our current cache interference channel detector supports the detection of secret-dependent loads/stores on a transient execution path. For all such memory accesses, an attacker can achieve *leakage* of the target (secret-dependent) address with a classic cache [20] or TLB [37] attack. Hence, to identify these loads/stores, we can simply report those that use a pointer tainted with the `secret` label during speculative emulation.

MDS detector. Our current MDS detector supports the detection of secret-accessing loads/stores on a transient execution path whose data can be leaked by an MDS exploit (i.e., sampling data via various CPU internal buffers, such as LFB [56], SB [10], etc.). We tested the (known) MDS triggering conditions and verified that an attacker can achieve *leakage* of data from arbitrary loads/stores on a mispredicted branch. Hence, to identify these loads/stores, we report cases where the pointer is under the control of the attacker during speculative emulation. These pointers are tainted with the `attacker` label and/or leading to unsafe loads/stores.

Port contention detector. Our current port contention detector supports the detection of secret-dependent branches on a transient path. For such branches, an attacker can *leak* a bit of the secret by issuing instructions that use the same execution units (i.e., ports) as the branch targets' instructions. Hence, to identify these branches, we report those that use a target tainted with a `secret` during speculative emulation. Note

```
if (a < size) {
    b = arr1[a]; // Transient out-of-bounds
    ↪  access loading memory massaged data
    c = arr2[ b ];
    d = arr3[ c ];
}
```

(a) Gadget where an attacker can inject data via memory massaging by taking advantage of a transient out-of-bounds read.

```
if (addr_is_mapped(ptr)) {
    x = *ptr; // Transient faulting accessing
    ↪  loading LVI-injected data
    y = arr2[ x ];
    z = arr3[ y ];
}
```

(b) Gadget where an attacker can inject data with LVI by taking advantage of a transient faulting load (see Listing 2e).

Listing 6: Gadgets that demonstrate the injection policies.

that future work could refine the set of reported port contention gadgets by using static analysis to determine whether a secret-tainted branch's possible targets are indeed SMoTHER-differentiable [9]—i.e., whether there is a sufficiently large enough timing difference generated by the port contention of one branch target to tell it apart from the port contention of another branch target.

B. Injection policies

Our unified *injection policies* combine our basic policy of tainting directly-controllable data injected via external input (e.g., syscall arguments) with our detection-based policies of tainting data injected via hardware/software vulnerabilities (e.g., memory massaging or LVI) which we refer to as indirectly-controllable attacker data. We distinguish between directly-controllable attacker data and indirectly-controllable attacker data because our *access policies* make use of this distinction, as explained later (Section VII-C). We refer to these labels as `attacker-dir` and `attacker-ind` (and `attacker` to refer to either one).

Injection Policy I: Directly-controllable data. *We taint all data which is directly-controllable from user space—that is, syscall arguments and the output of functions such as `get_user`, `copy_from_user`, and `strncpy_from_user`—with the `attacker-dir` label.*

Injection Policy II: Indirectly-controllable data. *We taint all data which is indirectly-controllable from an attacker, as modeled by our memory error and LVI detectors, with the `attacker-ind` label.*

Data injected via memory errors. We use our memory error detector to identify unsafe loads during speculative emulation; by default, upon detection and if the pointer is untainted, we add the `attacker-ind` label to the loaded value. Note that if the pointer is already tainted with an `attacker` label, such label is automatically propagated (pointer tainting enabled for loads). These policies are to model an attacker massaging controlled data in the target memory location. Since our memory error detector operates on heap and stack, these policies provide the attacker with plenty of exploitation strategies [13, 14, 61]. As an example, consider the gadget in Listing 6a which loads the attacker input by reading a heap object out-of-bounds, thereby possibly reading data from another object placed by the attacker using memory massaging. Attackers able to target heap memory massaging may gain indirect control over the data at `arr1[a]` and inject it into the otherwise-benign gadget.

Data injected via LVI. Similarly, we use our LVI detector to detect invalid loads during speculative emulation; by default, upon detection and if the pointer is untainted, we add the `attacker-ind` label to the loaded value (again existing `attacker` labels are propagated via pointer tainting). Consider the gadget in Listing 6b, which loads attacker-controlled data via a transitive faulty access, thereby retrieving a value from the store buffer. Attackers capable of LVI may gain indirect control over the data at `a->bar` and inject it into the otherwise-benign gadget.

In addition to tainting unsafe (including invalid) loads as `attacker-ind` data, we could also taint them as `attacker-ind` data loads whose pointer is tainted with an `attacker-dir` label. This is because if an attacker controls a pointer, then the attacker could hypothetically force it to load data through LVI or memory massaging. However, as explained later (Section VII-C), such loads are already tainted with the `secret` label and such modeling would be redundant (and lead to gadget over-reporting). We could also avoid using our detectors and only rely on pointer tainting, but this strategy leads to unnecessary false negatives (i.e., noncontrollable pointers still able to read memory massaged data or LVI data). Finally, we could disable pointer tainting on loads, but this strategy still misses cases of (safe/valid) loads where the attacker can control the loaded value.

C. Access policies

The raising of `attacker` labels to `secret` labels in access instructions is dependent on (1) the *type of control* the attacker has on the pointer, as determined by its taint sources, and (2) the *degree of control* the attacker has on the pointer, as approximated by the memory error detector.

Access Policy I: Raising directly-controllable attacker data. *If a load is unsafe and has a pointer with an `attacker-dir` label, then we add the `secret` label to the loaded value.*

We use both taint information and memory error detection to identify secret accesses. We observed that using only memory error detection leads to many false positives. This is because the attacker may not have enough control over the pointer to leak arbitrary data. Using only taint information, on the other hand, still leads to false positives, especially in the kernel. Indeed, the kernel contains many pointer masking operations (for input sanitization), which have the effect of keeping many controlled pointers always safe even in transient execution. Both strategies are an approximation of controllability: using only memory error detection but then flagging cases with fixed addresses as an indication for the lack of the attacker's control, or using only tainting but then flagging cases with limited controllability, as an indication that an

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x & MASK]; // In-bounds access
  z = arr2[y];
}
```

Listing 7: Gadget that is mitigated via a masking operation will not be falsely report as a gadget (see Listing 2d).

attacker will never be able to promote the access to go out-of-bounds. While state-of-the-art solutions [43, 47] have focused on these two extremes, we will later show there is no one-size-fits-all strategy and that different conditions require different treatments.

For example, consider the gadget in Listing 7, which is benign due to the masking operation which keeps the access in-bounds. In the gadget, we do not raise the `attacker-dir` label to a `secret` label (i.e., use only taint information) because it could never lead to an out-of-bounds access. In doing so, we avoid false positives which would arise from reporting harmless gadgets.

Access Policy II: Raising indirectly-controllable attacker data. *If a load has a pointer with an* `attacker-ind` *label, then we always add the* `secret` *to its output.*

Unlike the taint policies for raising directly-controllable data, we do not use memory error detectors in this case because indirectly-controllable data is generated (barring actual architectural bugs) within the same speculation window as the access instruction. Code paths using such transiently-injected data typically break global code invariants and we observed them to be rarely subject to pointer masking operations. As such, these paths offer almost unrestricted control to the attacker—unlike, say, syscall arguments and data loaded from usercopy functions, which kernel developers take efforts to sanitize against traditional exploits.

Note that the indirectly-controllable data which is loaded during the analysis is not generated by the user space attacker. It is either loaded from a code sanitizer's redzone or a similar dummy region (as described in Section VI). Hence, its possible values during analysis are limited, so any restriction based on a memory error detector would simply test against the values incidentally loaded at runtime, rather than any meaningful values injected by a user space attacker.

With this policy, we avoid false negatives which would arise from failing to identify an indirectly-controllable gadget because it incidentally did not lead to an unsafe *access* instruction.

D. Leakage policies

We use our hardware vulnerability detectors to identify gadgets that use cache-based, MDS-based, or port contention-based covert channels to leak `secret` information.

Leakage Policy I: Identifying cache-based gadgets. *If a memory access has a pointer with a* `secret` *value, then we report it as a cache-based gadget.*

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y]; // Leaks via cache
}
```

(a) Gadget leaking a secret via a cache-based covert channel.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x]; // Leaks via MDS
}
```

(b) Gadget leaking a secret via an MDS-based covert channel.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  if (y) {
    ... // Leaks via port contention
} }
```

(c) Gadget leaking a secret via a port contention-based covert channel.

Listing 8: Gadgets that demonstrate the leakage policies.

Hence, by propagating taint from `attacker`-controlled sources to dependent `secret` accesses, we find the simple Spectre-BCB gadget from Listing 1 by propagating taint as in Listing 8a. Similarly, if `*ptr` in Listing 2a is `attacker`-controlled, the policy also identifies the gadget.

Unlike *access* instructions, *leak* instructions require no memory error detector—only our simple cache interference detector. Such instructions will leave a trace on the cache (and TLB) regardless of whether it is, say, in-bounds or out-of-bounds. Hence, even if the leak instruction contains a masking operation to keep the index in-bounds—as in Listing 2b—we would still report it as a cache-based gadget, as it leaks at least *some* information. In doing so, we avoid false negatives which would arise from failing to identify gadgets that leave traces in the cache.

Leakage Policy II: Identifying MDS-based gadgets. *To identify MDS-based gadgets, we use the same policies as our access policies (which raise* `attacker` *data to* `secret` *data) with one minor difference (similar to our LVI policy): we do not report directly-controllable null-memory accesses because exploitation of such accesses is more difficult.*

For example, consider the MDS-based gadget in Listing 8b, where the errant access loads secret data into internal CPU buffers. Our policy reports an MDS gadget since the memory access outputs secret data, thereby potentially leaking the secret data to an attacker thread sharing an SMT core with the kernel thread. Note that for MDS-based gadgets, the *access* and *leak* steps occur in a single instruction.

Just as we avoid over-tainting harmless (e.g., in-bounds) *access* instructions, we avoid over-reporting MDS-based gadgets which are similarly benign, thereby avoiding a source of false positives.

TABLE III: Total lines of code (LOC) in the various components of KASPER. If a component is based on an existing one, the LOC given is the diff with respect to the original.

	LLVM pass	Runtime library
KSPECEM	964	2102
KDFSAN	251 (diff)	1975
TMANAGER	57	65
syzkaller	–	355 (diff)

Leakage Policy III: Identifying port contention-based gadgets. *If a* `secret` *affects the flow of execution—that is, if a branch's condition, switch's condition, indirect call's target, or indirect branch's targets is a* `secret`*—then we report it as a port contention-based covert channel.*

For example, consider the port contention-based gadget in Listing 8c. Our policy identifies the gadget because the secret determines whether the branch is taken—and in effect, determines the resulting port contention, which can leak a bit of the secret to an attacker that shares an SMT core with the kernel.

VIII. IMPLEMENTATION

Figure 4 presents the workflow of KASPER, our generalized transient execution gadget scanner for the Linux kernel. First, we build the kernel with KASPER support by using three components that each consist of a runtime library and an LLVM pass[2]: (1) Kernel Speculative Emulation Unit (KSPECEM), which emulates speculative execution due to a branch misprediction, (2) Kernel DataFlow Sanitizer (KDFSAN), which performs the taint analysis, and (3) Taint Manager (TMANAGER), which manages the vulnerability-specific taint policies. Next, we use a modified version of syzkaller [6] to fuzz the instrumented kernel and generate gadget reports. Finally, we calculate aggregate gadget statistics that aid developers in applying mitigations. Table III presents the total lines of code (LOC) in each of the main components of KASPER.

Kernel Speculative Emulation Unit. To simulate branch mispredictions, KSPECEM replaces branch conditions with a `SelectInst` that, depending on a runtime variable, will either take the original branch target or the inverse branch target. To simulate the resulting speculative execution, KSPECEM hooks store instructions so that any speculative memory write is reverted after speculative emulation finishes.

Kernel DataFlow Sanitizer. For our taint analysis engine, we ported the user space DataFlowSanitizer (DFSan [1]) from the LLVM compiler to the kernel. KDFSAN is, to our knowledge, the first general compiler-based dynamic taint tracking system for the Linux kernel. In contrast to the original DFSan implementation, we: (1) modified its shadow memory implementation to work in the kernel, (2) fixed its flawed label union operation (similar to existing work [12, 54]), (3) modified it to conservatively wash taint on the outputs of inline assembly, and (4) created custom taint wrappers to emulate the semantics of uninstrumentable code.

Taint Manager. TMANAGER implements the taint policies described in Section VII. It receives callbacks from the kernel and the KDFSAN runtime library and will e.g., apply taint to syscall arguments, or determine if a memory operation is an *inject*, *access*, or *leak* instruction. It re-uses checks from KernelAddressSanitizer (KASAN) [33] to determine whether a memory operation is an unsafe or an invalid access.

syzkaller. We used a customized version of syzkaller [6], an unsupervised coverage-guided fuzzer for the kernel, to maximize speculative code coverage. syzkaller's strategy of maximizing regular code coverage will inevitability maximize speculative code coverage since we start a speculative emulation window at every regularly-executed branch. We utilized qemu snapshots to begin every syzkaller testcase (a series of syscalls) from a fresh snapshot, avoiding leftover taint from previous testcases.

Gadget aggregation. After fuzzing, KASPER parses the execution log for gadget reports. Then, it filters out duplicate reports, categorizes them by type, and prioritizes them based on a set of exploitability metrics (see Appendix C for a description of these metrics). Finally, it stores the resulting aggregate gadget statistics into a database that is used by a web interface to present the results. In Appendix E, we present the interface that allows kernel developers to easily process the found gadgets.

IX. EVALUATION

We evaluate KASPER's efficacy compared to existing solutions and KASPER's gadgets found in the Linux kernel. We perform our evaluation on an AMD Ryzen 9 3950X CPU with 128GB of RAM, where the KASPER-instrumented kernel (v5.12-rc2) runs as a guest VM on a host running Ubuntu 20.04.2 LTS (kernel v5.8.0).

A. Comparison with previous solutions

We compare KASPER against previous approaches in a variety of ways. First, as a micro-benchmark, we evaluate KASPER and all other approaches on the Kocher gadget dataset [31]. Then, as a macro-benchmark, we evaluate KASPER and other dynamic approaches on the syscalls invoked by UNIX's `ls` command. Finally, in Appendix B, we evaluate the performance of KASPER and compare it to existing approaches, where applicable.

Micro-benchmark. We evaluate KASPER and previous work against Paul Kocher's 15 Spectre examples [31], which were originally designed to evaluate the effectiveness of the Spectre mitigation in MSVC (Microsoft Visual C++). Although the examples are simple—as gadgets in real-world programs are often much more complex—they represent one of the few well-defined microbenchmarks available for direct comparison with different solutions.

Table IV shows the results for tools that are based on static and dynamic analysis. Static analysis tools specifically model the Spectre-PHT gadgets, combining the access to the secret

[2] We opted for an LLVM-based implementation due to its low complexity and better performance compared to e.g., a full-system emulator. Future work may see improvements by complementing KASPER with a binary-based approach since it does not require special care of low-level code.

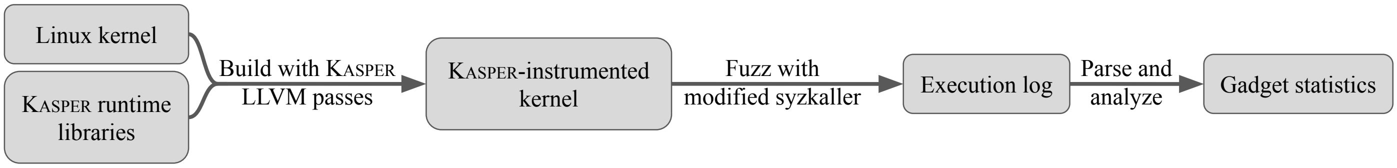

Linux kernel
Kasper runtime libraries
Build with Kasper LLVM passes
Kasper-instrumented kernel
Fuzz with modified syzkaller
Execution log
Parse and analyze
Gadget statistics

Fig. 4: The workflow of Kasper, which takes a vulnerable Linux kernel, identifies gadgets, and finally presents statistics to aid developers in applying mitigations.

TABLE IV: Gadgets detected in the 15 Spectre Samples Dataset [31] by various solutions.

		Accesses detected	*Leaks* detected
Static	MSVC [44]	–	7
	RH Scanner [15]	–	12
	oo7 [59]	–	15
	Spectector [22]	–	15
Dynamic	SpecFuzz [43]	15	0[a]
	SpecTaint [47]	15	14[b]
	Kasper (User/Cache-only)	15	14
	Kasper (User/Cache/Port-only)	15	15

[a] SpecFuzz’s eval. reports 15 leaks since it assumes all *accesses* are *leaks*.
[b] SpecTaint’s eval. reports 15 leaks since it assumes arbitrary ptrs. are secret.

and its leakage through a covert channel. While these tools scale to such small code snippets, they often miss many cases with more complex gadgets combining multiple attack vectors. Furthermore, it is difficult to run solutions based on symbolic execution [22] on large codebases such as the Linux kernel without losing soundness.

In the dynamic analysis category, SpecFuzz [43] depends on address sanitizers to detect invalid accesses. Without data flow analysis, tracking the data dependency between accessing and leaking of a secret is not possible. Assuming the second access encoding the secret in the reload buffer is inbounds, SpecFuzz is not able to detect it. SpecTaint [47] makes use of dynamic taint analysis, allowing it to detect dependencies between the access and leakage. While the authors report that they detect all 15 variants, it is unclear how they detect variant v10, that leaks the secret through the outcome of the branch. In direct communication with the authors, they mentioned that SpecTaint assumes that the leaking pointer is tainted as a secret. However, this makes it independent of the presented variant.

Without implicit flow tracking, Kasper detects the access of the secret for all 15 variants and the transmission of 14 of those. For the undetected variant (v10), Kasper’s port contention policies detect the attacker-controlled branch giving the attacker the ability to leak the secret by controlling the outcome of the branch.

Macro-benchmark. To provide a more realistic scenario, we evaluate Kasper and previous work by running the `ls` UNIX command. Since SpecFuzz [43] and SpecTaint [47] are only implemented for user space programs, we adapted Kasper to model their functionality for the kernel. To model SpecFuzz, we report any address sanitizer violation within speculative emulation as a Cache gadget. To model SpecTaint[3], we taint syscall arguments and user copies as `attacker` data, taint the output of any `attacker`-dependent speculative load as `secret` data, and report any `secret`-dependent access as a Cache gadget.

TABLE V: Cache gadgets reported in the kernel by various solutions when running the `ls` UNIX command.

	Total Cache gadgets reported	FP rate	FN rate (User-only)	FN rate
SpecFuzz [43]	662	99%	33%	60%
SpecTaint [47]	688	99%	0%	40%
Kasper (User-only)	8	25%	0%	40%
Kasper	14	29%	0%	0%

For each solution, Table V presents: (1) the total number of Cache gadgets reported, (2) the false positive rate, (3) the false negative rate when scanning for gadgets controlled only by User input, and (4) the false negative rate when scanning for gadgets controlled by User, Massage, and LVI input. In the scenario where an attacker can only *inject* User input, we define a true positive as a gadget that: (1) *injects* directly-controllable attacker data (i.e., syscall arguments or user copies), (2) *accesses* a secret by dereferencing a pointer that is both unsafe (i.e., out-of-bounds or use-after-free) and directly-controllable (i.e., flowing from the *injected* data), and (3) *leaks* the secret by dereferencing a pointer that is secret (i.e., flowing from the *accessed* data). In the scenario where an attacker can also *inject* Massage and LVI data, we define a true positive as a gadget that may additionally: (4) *inject* indirectly-controllable attacker data by dereferencing a pointer that is either invalid (i.e., a non-canonical address or a user address with SMAP on) or unsafe, and (5) *access* a secret by dereferencing a pointer that is indirectly-controllable.

As shown in the table, existing solutions’ pattern-based approaches incur a high FP rate (99%) and a substantial FN rate (up to 33%). In contrast, Kasper (User-only)’s more principled approach drastically decreases the FP rate (25%) and matches the best case FN rate (0%). However, when considering gadget primitives beyond those in the BCB pattern—i.e., Massage and LVI attacker inputs—FN rates for existing approaches (and for the limited version of Kasper) increase significantly (to 40%–60%). Since the full version of Kasper models these primitives (and more), it has no FNs and maintains an acceptable FP rate (29%). We observed similar results on the GNU core utilities other than `ls`.

[3] We base our implementation of SpecTaint on the information provided in the paper, because while the paper states the authors’ intention to release SpecTaint as open source, the code is not yet available.

TABLE VI: Different gadgets discovered by Kasper.

Gadget type	Number of reports
PHT-User-Cache	147
PHT-Massage-Cache	47
PHT-LVI-Cache	12
Total PHT-*-Cache	183
PHT-User-MDS	600
PHT-Massage-MDS	193
Total PHT-*-MDS	722
PHT-User-Port	407
PHT-Massage-Port	123
Total PHT-*-Port	474
Total PHT-*-*	1379

We verified our FP/FN rates by checking whether each reported gadget satisfies our definition of a TP. First, we verified that Kasper's FPs are due to overtainting; moreover, these FPs are also nondeterministic (i.e., they only appear in specific runs). Second, we verified that all of SpecFuzz's FPs are due to its inability to track attacker/secret data, causing it to report *leaks* of non-secret data (as in Listing 2c). Third, we verified that all of SpecFuzz's (User-only) FNs are due to its inability to identify in-bound *leaks* (as in Listing 2b). Fourth, we verified that all of SpecTaint's FPs are due to its inability to filter out safe *accesses* (as in Listing 2d). Finally, we verified that all remaining FNs are due to existing approaches (and the limited version of Kasper) not modeling Massage or LVI attacker input.

From these results, we can conclude that previous approaches are insufficient for a variety of reasons. First, the high FP rates for existing techniques are problematic because hardening every reported gadget would lead to a substantial slowdown, yet attempting to verify all the reported gadgets is impractical. Second, SpecFuzz's substantial FN rate is problematic because it leaves many unidentified gadgets vulnerable. Finally, the high FN rates when considering Massage and LVI primitives—which would be far worse if also considering MDS and Port primitives—are problematic because they highlight how previous approaches leave generic gadgets completely vulnerable.

B. Gadgets found in the kernel

We fuzzed the kernel for 18 days with 16 virtual machines running in parallel and report the number of gadgets found in Table VI. Kasper currently taints data copied from userspace (User), data vulnerable to memory massaging (Massage) and data vulnerable to LVI-injection (LVI) as `attacker` input. Furthermore, the prototype assumes that `secret` data can leak either through microarchitectual buffers (MDS), port contention (Port) or cache-based covert channels (Cache). Although Kasper can model PHT-LVI-MDS and PHT-LVI-Port gadgets, we do not report such variants since we were unable to exploit them in practice (see Section IV).

Most of the gadgets found by Kasper allow information to leak via MDS after a PHT misprediction. The input for these gadgets mostly comes from syscall arguments or values that the attacker can indirectly control in memory via massaging techniques. We present a case study of one such gadget that is difficult to mitigate in Section XI. Kasper also finds 147 gadgets with the same capabilities as Spectre-BCB, which are still missed by existing mitigations due to limitations in the kernel's static analysis-based gadget scanning tools, which only look for specific patterns.

X. Limitations

Although we have shown that Kasper outperforms state-of-the-art gadget scanners, it is still limited relative to the *ideal* gadget scanner—i.e., a scanner that detects *practically exploitable* gadgets with *perfect precision*.

Practical exploitability. We do not evaluate the practical exploitability of every gadget found. Such an evaluation is infeasible in bounded time: it would require testing each gadget against a myriad of possible microarchitectures, attack patterns to facilitate the exploit (e.g., concurrent cache evictions [20]), etc. Instead, like existing kernel Spectre mitigations and state-of-the-art gadget scanners [43, 47], we aim to provide a more comprehensive security-by-design, since even gadgets that are seemingly nonexploitable now may become practically exploitable due to seemingly-innocuous changes—e.g., microcode updates, code refactors, or different compiler versions [30]. Nonetheless, in Section XI, we present a proof-of-concept exploit of a gadget found by Kasper. Furthermore, in Appendix C, we discuss heuristics that developers can use to prioritize gadgets that are more likely to be exploitable in practice.

Completeness. Kasper's results may be incomplete for a couple of reasons. First, similar to existing dynamic gadget scanners [43, 47], we inherit dynamic analysis's inherent limitation of incomplete coverage. In Appendix A, we evaluate this limitation and conclude that as fuzzing progresses, FNs due to incomplete coverage become more and more rare. Second, since (K)DFSan does not track attacker-controllable implicit data flows, Kasper will fail to identify gadgets that rely on them. However, these FNs may be less useful to an attacker, because implicit flows normally propagate only one bit of data.

Soundness. Similar to existing scanners based on dynamic taint analysis (DTA) [47], we inherit DTA's inherent soundness limitations. Specifically, since DTA cannot model data-flow constraints (e.g., the effect of arbitrary arithmetic or bitwise operations on data), Kasper may report gadgets that are not entirely controllable by an attacker. For example, even though it might only be possible for an `attacker`-controllable *access* to go one byte out-of-bounds—rather than *arbitrarily* out-of-bounds—Kasper will overlook this, and still taint the output as a `secret`. To mitigate these FPs, Kasper washes taint for common data masking operations that are part of mitigations in the Linux kernel (i.e., `array_index_nospec`), but a general solution remains out of the scope of DTA. A generic way to address this problem is to rely on symbolic (or concolic) execution, but state-of-the-art techniques can only scale to a limited number of basic blocks of kernel execution [65]. In contrast, Kasper can scalably find gadgets with attacker-controlled data propagating even across multiple syscalls.

Fidelity. Since our implementation does not faithfully model every aspect of a natively-run kernel, its results may be imprecise. First, we cannot precisely model nondeterminism—e.g., from timer interrupts occurring at different program

```
#define list_for_each_entry(pos, head, member)
for (pos = list_first_entry(head,
           typeof(*pos), member);
     &pos->member != (head);
     pos = list_next_entry(pos, member))
```

Listing 9: Linux implementation for generic list iterations. pos is the current list element, head is the head of the list, and pos->member is the next list element.

states in the KASPER kernel compared to the native kernel. We can mitigate any resulting FNs by increasing coverage. Second, we do not precisely model all microarchitectural details—e.g., exact speculative window sizes, pipeline stalls caused by memory aliasing, etc. We can mitigate any resulting FNs by extending KASPER to model even more low-level microarchitectural details. We consider any fidelity-related FPs to be acceptable, as described above (i.e., seemingly-innocuous changes may turn a FP into a TP).

XI. Case Study

We have shown that KASPER finds a wide range of gadgets in the hardened Linux kernel codebase. To demonstrate that the presence of such gadgets is serious, we now present a case study for one of the (unmitigated) gadgets found by KASPER. It shows that focusing on simple gadgets is insufficient and mitigation can be far from trivial. Finally, we analyze the exploitability of the found gadget. We refer the interested reader to Appendix D for an additional case study illustrating the need for a dynamic analysis tool for the Linux kernel.

A. *List implementation of the kernel*

Our case study is fundamental to the (double linked) list implementation that is used pervasively in the Linux kernel. The kernel's list implementation is cyclic and consists of a head element (of type list_head) which is typically a field in another data structure, and list elements containing the data, where the last element points back to the head element. A data structure can become a list element, if it also embeds the list_head struct as one of its fields. The list_head simply contains pointers to the list_head fields in the previous and next list elements (or the head).

To iterate over a list, the kernel provides macros such as list_for_each_entry, shown in Listing 9. Here pos points to the data structure and pos->member to the embedded list_head. The list iteration terminates when it reaches the head element in the cyclic list. List iterators are also used pervasively in the kernel codebase with more than 2600 uses.

B. *A list_for_each_entry gadget in keyring.c*

The security implications of the list_for_each_entry implementation become clear when we consider the gadget found by KASPER in the find_keyring_by_name function in keyring.c. Listing 10 shows the relevant code snippet. Simulating a branch misprediction, KASPER flips the terminating condition of the list iterator (&pos->member != head), resulting in an additional iteration with &pos->member == head. Note that when the list iteration is speculatively executed for the head element, there is no associated key data structure. In other words, keyring and &keyring->user point to out-of-bounds memory, in this case belonging to the data structure containing the head element. The type confusion results in the access to keyring->user->uid dereferencing an out-of-bounds read pointer.

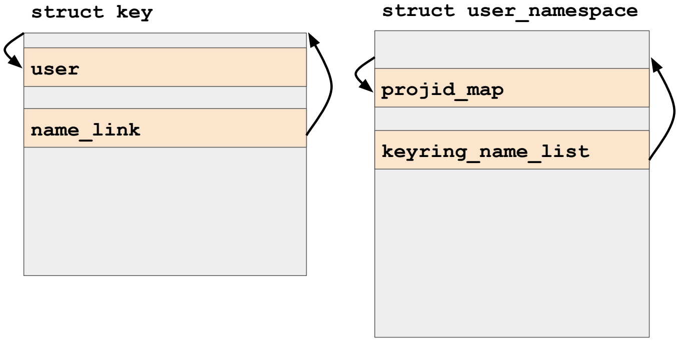


Fig. 5: Through the type confusion head is assumed to be within a key struct instead of the user_namespace struct.

KASPER not only detects the KASAN violation, but further reports the gadget as capable of leaking secrets trough MDS, as it verifies through its dynamic taint analysis that the pointer is attacker controlled.

C. *Exploitation*

First, we evaluate the controllability of the out-of-bounds read pointer. The type-confusion assumes that &ns->keyring_name_list is within a key struct, but in reality the head element is in a user_namespace struct. To compute the location of &keyring->user in the speculative iteration, we first compute the offset of name_link within the key struct. Next we compute the offset of user in the struct and apply those offsets to &ns->keyring_name_list as shown in Figure 5. The out-of-bounds read pointer keyring->user is read from ns->projid_map which can be easily controlled from user space through the proc interface.

We evaluated the gadget on an i7-7700K machine with a recent v5.12 Linux kernel. First, we verified whether the branch can be mistrained. Since the attacker can control the length of the keyring_name_list list by adding keys through the add_key, mistraining the branch condition in-place is not difficult. Moreover, for easier exploitation, the gadget can be reached close to the system call entry of keyctl with the KEYCTL_JOIN_SESSION_KEYRING operation.

We build a simple Flush+Reload [62] proof of concept to verify that the attacker-controlled pointer is actually used in the speculative load operation. For verification purposes, we disabled SMAP and set the pointer pointing directly into the reload buffer and observed the signal by executing the keyctl system call between flushing and reloading. We confirm that the attacker-controlled pointer is dereferenced during speculative execution.

In principle, any value loaded by a speculative load can be leaked cross-thread with MDS [10, 51, 56] if SMT is enabled. Since SMT is not disabled by default in the latest version

```
struct key *find_keyring_by_name(const char *name, bool uid_keyring) {
  struct user_namespace *ns = current_user_ns();
  struct key *keyring;
  ...
  list_for_each_entry( keyring , &ns->keyring_name_list, name_link) {
    if (!kuid_has_mapping(ns,  keyring->user->uid ))
      continue;
  ...
} }
```

Listing 10: PHT-MASSAGE-MDS gadget in `find_keyring_by_name`.

```
while(1)
  syscall(__NR_keyctl,
      KEYCTL_JOIN_SESSION_KEYRING,
      "kasper", 0, 0, 0);
```

(a) Attacker thread that repeatedly invokes the vulnerable `keyctl` system call.

```
while(1) {
  madvise(leak+4096, 4096, MADV_DONTNEED);
  flush(reloadbuffer);
  asm volatile(
    "movdqu (%0), %%xmm0    \n"
    "movq %%xmm0, %%rax     \n"
    "andq $0xff, %%rax      \n"
    "shl $0xa, %%rax        \n"
    "prefetcht0 (%%rax, %1) \n"
    "mfence                 \n" ::
    "r"(leak + 4096 - 14),
    "r"(reloadbuffer):"rax","rbx","rcx");
  reload(reloadbuffer, results);
}
```

(b) Attacker thread that repeatedly encodes in-flight secret data into a `reloadbuffer`.

Listing 11: Simplified proof of concept exploit consisting of two simultaneously executing threads.

of Linux, it is still vulnerable to MDS when leaking from the sibling hyperthread. We verify this with a simple proof of concept where one thread repeatedly executes the `keyctl` system call and the other thread reads in-flight data and encodes it within a Flush+Reload buffer. We use `madvice` to force a page fault across a page boundary which is required to leak the in-flight data. Listing 11 presents a simplified version of the proof of concept. Without synchronization between the two threads (which was not the focus of our research), the signal strength depends on the the leaking load in the system call occurring roughly at the same time as as the load from CPU-internal buffers in the reading thread. We verified that a signal exists if the loads are happening approximately at the same time in both threads, leaking the secret from a kernel buffer to user space.

Mitigating the presented gadget is far from trivial. Adding a spot mitigation in `keyring.c` leaves all other uses of `list_for_each_entry` vulnerable. Mitigating the list iterators, that are frequently used throughout the kernel, may cripple performance when using `lfence` in every iteration of the loop. Other kernel mitigations such as `array_index_nospec` are not applicable with the current list implementation.

XII. RELATED WORK

Checkpointing for transactional execution. While checkpointing was used to recover from software faults [35, 57], we require such high frequency checkpointing (on every conditional branch) that we built a checkpointing mechanism for the Linux kernel from scratch. However, the solution is general and applicable to other scenarios (e.g., crash recovery).

Taint tracking in the kernel. For KASPER, we built KDFSAN, the first generic compiler-based dynamic taint tracking system for the kernel. Other non-generic, non-compiler-based, and static taint tracking systems have also found bugs in the kernel. For example, KMSAN is a widely-used compiler-based dynamic taint tracking system targeted at detecting uninitialized memory usage; it has found hundreds of bugs in the Linux kernel [46]. Furthermore, Bochspwn Reloaded is an emulation-based dynamic taint tracking system targeted at detecting uninitialized memory disclosures to user-mode; it has found 70 bugs in the Windows kernel and 10 bugs in the Linux kernel [29]. Finally, a wide range of static taint tracking systems have found numerous bugs in the kernel [28, 39, 60].

Speculative gadget scanners. Shortly after the publication of Spectre [32], the Red Hat Spectre V1 scanner [15] and Microsoft's Visual C/C++ Compiler [44] tried finding and mitigating Spectre V1 gadgets using well defined code patterns. More advanced static analysis gadget scanners followed. oo7 [59] uses binary-level static taint analysis and has analysis times of 74 hours on average on medium sized user applications, scaling it to large codebases such as the Linux kernel is impractical. SPECTECTOR [22] and KLEESPECTRE [58] use symbolic execution to find Spectre based information leaks. Without sacrificing soundness and completeness, symbolic execution becomes a bottleneck when scaling to large real-world codebases like the kernel [65]. The Linux kernel currently relies on manual analysis and static analysis tools such as smatch [11] to identify potential Spectre gadgets. However, this still involves manual inspection because found code locations are patched at the source code level and it suffers of a high rate of false positives. SpecFuzz [43], a dynamic analysis tool, combines fuzzing with the use of sanitizers to detect and mitigate potential speculative memory leaks within user applications. SpecTaint [47] uses dynamic taint analysis to track attacker controllability and leaking of the secret. Similar to SpecFuzz, it targets user space programs which rarely have mitigations applied.

XIII. Conclusion

We presented Kasper, a solution for finding generalized transient execution gadgets in the Linux kernel. Where existing gadget scanners limit themselves to specific Spectre patterns (in user programs), Kasper abstracts away pattern-specific details and instead models the essential steps of an attack: injecting controllable data, accessing a secret, and then leaking the secret. Moreover, where existing scanners target only the primitives used in the BCB pattern, Kasper models the wide variety of primitives that are at an attacker's disposal. As a result, Kasper finds gadgets that are well out of reach of existing techniques. We conclude that current Spectre mitigations in the Linux kernel are wholly insufficient.

Disclosure. We disclosed our findings to the Linux kernel and to Intel on February 11, 2021. We shared our paper, gadgets found, and patches for the discussed gadgets (Section XI and Appendix D) with the affected parties, who acknowledged our findings and agreed to a public disclosure date of Jan 25, 2022.

Acknowledgements

We thank the anonymous reviewers for their valuable comments, Felix Goosens for verifying a signal on an LVI-MDS gadget, and Alexander Potapenko for developing KMSAN, which parts of KDFSan are based on. This work was supported by the EU's Horizon 2020 research and innovation programme under grant agreement No. 825377 (UNICORE), Intel Corporation through the Side Channel Vulnerability ISRA, the Netherlands Organisation for Scientific Research through projects "TROPICS", "Theseus", and "Intersect", and EKZ through project "VeriPatch". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] "DataFlowSanitizer," https://clang.llvm.org/docs/DataFlowSanitizer.html.

[2] "Kasper," https://www.vusec.net/projects/kasper.

[3] "LibHTP," https://github.com/OISF/libhtp.

[4] "OpenSSL," https://www.openssl.org.

[5] "syzbot," https://syzkaller.appspot.com.

[6] "syzkaller," https://github.com/google/syzkaller.

[7] "The Linux Kernel documentation: MDS - Microarchitectural Data Sampling," https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html.

[8] "The Linux Kernel documentation: Spectre Side Channels," https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html.

[9] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019.

[10] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-Resistant CPUs," in *CCS*, 2019.

[11] D. Carpenter, "Smatch check for Spectre stuff," https://lwn.net/Articles/752409, 2018.

[12] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *S&P*, 2018.

[13] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities," in *USENIX Security*, 2020.

[14] Y. Chen and X. Xing, "SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel," in *CCS*, 2019.

[15] N. Clifton, "SPECTRE Variant 1 scanning tool," https://access.redhat.com/blogs/766093/posts/3510331, 2018.

[16] J. Corbet, "The current state of kernel page-table isolation," https://lwn.net/Articles/741878, 2017.

[17] D. E. Denning, "A Lattice Model of Secure Information Flow," *SOSP*, 1975.

[18] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *TOCS*, 2014.

[19] J. Fustos, M. Bechtel, and H. Yun, "SpectreRewind: Leaking Secrets to Past Instructions," in *ASHES*, 2020.

[20] E. Goktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative Probing: Hacking Blind in the Spectre Era," in *CCS*, 2020.

[21] "Open Source Security Inc. Announces Respectre: The State of the Art in Spectre Defenses," https://grsecurity.net/respectre_announce, 2018.

[22] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled Detection of Speculative Information Flows," in *S&P*, 2020.

[23] J. Horn, "Reading privileged memory with a side-channel," https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018.

[24] ——, "speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2019.

[25] Intel, "Retpoline: A Branch Target Injection Mitigation," 2018.

[26] ——, "Speculative Execution Side Channel Mitigations," 2018.

[27] ——, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 2020.

[28] R. Johnson and D. Wagner, "Finding User/Kernel Pointer Bugs With Type Inference," in *USENIX Security*, 2004.

[29] M. Jurczyk, "Bochspwn Reloaded: Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking," in *Black Hat USA*, 2017.

[30] O. Kirzner and A. Morrison, "An Analysis of Speculative Type Confusion Vulnerabilities in the Wild," in *USENIX Security*, 2021.

[31] P. Kocher, "Spectre Mitigations in Microsoft's C/C++ Compiler," https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[32] 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 *S&P*, 2019.

[33] A. Konovalov and D. Vyukov, "KernelAddressSanitizer (KASan): a fast memory error detector for the Linux kernel," *LinuxCon North America*, 2015.

[34] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *WOOT*, 2018.

[35] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, "Hypervisor-assisted Application Checkpointing in Virtualized Environments," in *ASPLOS*, 2011.

[36] 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*, 2018.

[37] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "DOLMA: Securing Speculation with the Principle of Transient Non-Observability," in *USENIX Security*, 2021.

[38] K. Lu and H. Hu, "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in *CCS*, 2019.

[39] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers," in *USENIX Security*, 2017.

[40] G. Maisuradze and C. Rossow, "Ret2Spec: Speculative Execution Using Return Stack Buffers," in *CCS*, 2018.

[41] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," in *POPL*, 1999.

[42] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, "You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass," *arXiv preprint arXiv:1805.08506*, 2018.

[43] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing Spectre-type vulnerabilities to the surface," in *USENIX Security*, 2020.

[44] A. Pardoe, "Spectre mitigations in MSVC," https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc, 2018.

[45] C. Percival, "Cache missing for fun and profit," 2005.

[46] A. Potapenko, "Add KernelMemorySanitizer infrastructure," https://lwn.net/Articles/878652, 2021.

[47] Z. Qi, Q. Feng, Y. Cheng, M. Yan, P. Li, H. Yin, and T. Wei, "SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets," in *NDSS*, 2021.

[48] 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 *USENIX Security*, 2021.

[49] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *J-SAC*, 2003.

[50] S. Schumilo, C. Aschermann, A. Abbasi, S. Worner, and T. Holz, "NYX: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types," in *USENIX Security*, 2021.

[51] 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.

[52] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *ESORICS*, 2019.

[53] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "ASSURE: Automatic Software Self-Healing Using Rescue Points," in *ASPLOS*, 2009.

[54] E. Sultanik, "Two New Tools that Tame the Treachery of Files," https://blog.trailofbits.com/2019/11/01/two-new-tools-that-tame-the-treachery-of-files, 2019.

[55] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *S&P*, 2020.

[56] 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*, 2019.

[57] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Lightweight Memory Checkpointing," in *DSN*, 2015.

[58] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, "KLEESPECTRE: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution," *TOSEM*, 2020.

[59] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead Defense against Spectre Attacks via Program Analysis," *TSE*, 2021.

[60] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving Integer Security for Systems with KINT," in *OSDI*, 2012.

[61] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel," in *CCS*, 2015.

[62] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.

[63] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *MICRO*, 2019.

[64] P. Zijlstra, "Add static_call()," https://lwn.net/Articles/824406, 2020.

[65] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel," in *USENIX Security*, 2021.

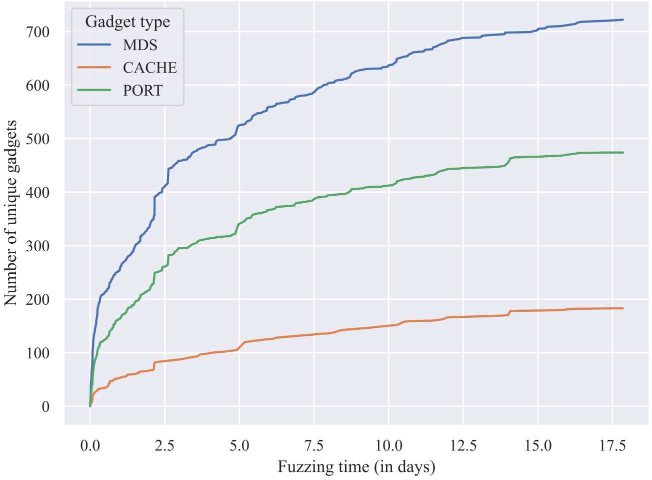


Fig. 6: Gadgets found by KASPER over time, separated by gadget type.

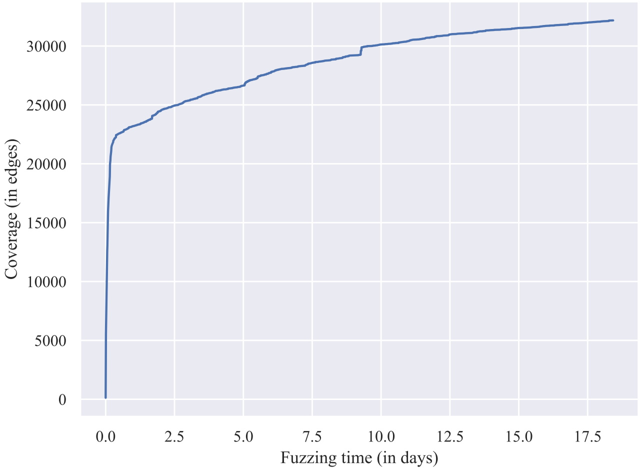


Fig. 7: Covered edges in the Linux kernel reported by syzkaller.

APPENDIX A
COVERAGE EVALUATION

We evaluate the completeness of KASPER's fuzzing results (Section IX-B) based on its coverage in both normal execution and speculative emulation.

A. *Normal execution coverage*

First, we find that KASPER's total gadgets found (Figure 6) begins to flatten at around 18 days; this is not surprising, since we fuzzed the kernel until KASPER found only 3 new gadgets per day (out of 1379 gadgets found in total). Next, we find that

TABLE VII: Causes for rollback in speculative emulation.

Rollback cause	Basic	+ Page fault suppression	+ Inline assembly patches
Max spec length	20.8%	22.9%	56.9%
Return	32.9%	33.8%	39.7%
Inline asm	38.4%	41 %	0.2%
Indirect calls	0.8%	1.2%	1.3%
Interrupts	6.5%	0.5%	1.2%
Blacklisted function	0.6%	0.6%	0.7%

Kasper's code coverage (Figure 7) also flattens, confirming that further fuzzing will most likely execute already-executed code, and as a result, uncover fewer and fewer gadgets.

To evaluate how long it would take for Kasper's code coverage to *completely* flatten (and in effect, uncover almost *all* gadgets), we compare Kasper's code coverage to an uninstrumented kernel's code coverage. We fuzzed an uninstrumented kernel for 18 days and found that in the final 24 hours, the baseline's coverage increased by only 0.15%, and in the same span, Kasper's coverage similarly increased by only 0.61%. In other words, the baseline's coverage—like Kasper's coverage—flattens, but not completely. Since even the baseline does not completely flatten in bounded time[4], we cannot expect Kasper to completely flatten in bounded time. Hence, future work on improving syzkaller's coverage would not only improve the completeness of Kasper's results—it would also improve *all* kernel fuzzing results.

B. Speculative emulation coverage

Apart from *normal* code coverage, KSpecEm specifically targets increasing the code coverage in the *speculative window*. Simply executing basic blocks within speculative emulation is not sufficient. Ideally, KSpecEm should execute a basic block speculatively in every speculative window that may cover it. In other words, we should exhaust speculative execution windows as much as possible and eliminate premature rollbacks.

Lightweight checkpointing in the kernel is more challenging than in well-defined user space programs, due to the many complex operations, frequent use of inline assembly, interactions with device memory, and complex control flow resulting from interrupts. As described in Section VI-B, to handle such cases we allow speculative emulation to stop gracefully to ensure memory integrity on a rollback. Stopping speculative emulation in the above cases limits the length of the emulation window.

Table VII presents the different causes of rollbacks in three scenarios: the basic implementation of KSpecEm and two improvements that we later made to improve speculative code coverage. These numbers report the causes for rollbacks averaged over all instrumented functions when executing the `ls` command. A negligible amount of rollbacks (3.4%) are due to limitations introduced by unique challenges of the Linux kernel, as explained earlier in Section VI-B. We set the maximum of speculative instructions for all evaluations to 250 executed LLVM IR instructions, which is in line with the size of the ROB in modern microarchitectures [27] and what has been used in recent work (for x86 instructions) [43, 47]. We have analyzed the ratio of the number of LLVM IR instructions to x86 instructions within the Linux kernel per function using LLVM passes. The median of the ratio is 0.92, concluding that counting LLVM IR instructions is a reasonable approximation for the number of machine code instructions. The first row shows the percentage of cases where KSpecEm rolls back because we reach this maximum speculation length. Ideally, the percentage of rollbacks due to reaching the maximum speculation length should be as high as possible.

We see that a small percentage of rollbacks can be attributed to indirect calls where the target address cannot be verified at compile time, interrupts, and blacklisted functions such as those that interact with I/O memory. This proves that, even for drivers code, we have isolated the small locations interacting with I/O mapped memory, still reaching high speculative code coverage within those kernel components. The low number of rollbacks due to interrupts show that these interrupts, caused by exceptions creating an unrecoverable state, are not a significant limitation of Kasper. A major source of rollbacks consists of returns as speculative emulation tries to go up in the call trace from the start of the checkpoint. In contrast, returning in functions that have been called within speculative emulation is safe since it will return back into a function that was already executed within emulation. Future work can support additional returns by keeping track of safe functions within the call trace, and thus increase speculative code coverage even further.

In the basic implementation (column 2), we stop on all interrupts (including exceptions and timers) and every occurrence of inline assembly. In this case, speculative emulation reaches the maximum speculation length in a modest 20.8% of all cases. However, by adding Page Fault suppression (column 3), we find new classes of attack (e.g., LVI) and reduce interrupts from 6.5% to a mere 0.5%, to arrive at 22.9% of the cases that exhaust the maximum speculation window. Moreover, by modeling the most frequently-used inline assembly fragments, we increase the percentage of rollbacks due to reaching the speculation length to 56.9%. As shown by our case studies, these improvements enabled Kasper to find a wide range of gadgets.

Appendix B
Performance Evaluation

We evaluate the performance of Kasper relative to an uninstrumented kernel and where possible, relative to previous approaches.

Analysis time. Similar to previous work [43], we evaluate the time overhead of our approach based on the fuzzing throughput (i.e., the number of testcases over time). First, we compare the fuzzing throughput of Kasper against the uninstrumented kernel. Next—since we observed that our modifications to syzkaller, which use qemu's snapshot feature to revert taint between testcases (see Section VIII), introduced a major overhead—we also compare the fuzzing throughput of Kasper against the uninstrumented kernel running with our modified version of syzkaller.

[4] Indeed, Google runs syzkaller continuously on many different kernel versions and configurations, and even its longest-running fuzzing campaigns continue to gain code coverage [5].

We ran each setup for 36 hours on the same machine used for our fuzzing evaluation (Section IX-B). We found that on average, KASPER executes 136 testcases per hour, compared to the uninstrumented kernel executing 45,933 per hour; however, when running with our modified version of syzkaller, the uninstrumented kernel executes a mere 252 testcases per hour. Hence, we can attribute a *1.8x slowdown due to KASPER's instrumentation* and a *183x slowdown due to our syzkaller modifications*. Since our modifications to syzkaller were not the focus of this work, we consider this an acceptable overhead; orthogonal (and concurrent) efforts to optimize qemu's snapshot feature can be used to improve the throughput [50]. Furthermore, note that reverting taint between testcases is not strictly necessary for KASPER. Nonetheless, we opted for this strategy because we prefer to have reproducible results (by starting each testcase from a clean snapshot) over quick results (by fuzzing without snapshotting).

For comparison, SpecFuzz reports a *23x slowdown in the best case* (on libHTP [3]) and a *560x slowdown in the worst case* (on OpenSSL [4]). Hence, relative to the 1.8x overhead of KASPER's instrumentation, SpecFuzz's instrumentation incurs a sizable overhead. We attribute this difference to SpecFuzz's use of nested speculation to improve speculative code coverage: i.e., SpecFuzz inverts many conditional branches within a single speculative emulation window, whereas KASPER only inverts one. Although SpecFuzz observes that most gadgets found are within the lower orders of nested speculation (i.e., only requiring one or two inverted branches), nested speculation (as well as other forms of speculation) can be integrated in KASPER in future work.

Unfortunately, we cannot meaningfully compare against SpecTaint for a couple of reasons. First, its performance numbers are not relative to a tangible baseline: e.g., it does not present baseline times for programs when *not* run with SpecTaint; also, it does not define when an analysis is "finished", even though the analysis times it presents use this metric. Second, the code is not yet available, so we cannot reproduce its performance numbers. However, we estimate that, because it uses a full-system emulation-based approach, it likely incurs a more significant overhead compared to KASPER's and SpecFuzz's LLVM-based approaches.

Memory consumption. KASPER consumes just over 4x memory relative to a baseline system. Of this overhead, 2x is required by syzkaller for its snapshot feature. Another 2x is required by KDFSAN's shadow memory, which allocates a page of shadow memory for every page of kernel memory. Beyond that, a constant, negligible amount of memory is required for KSPECEM's tracking of speculative memory writes and other internal data structures. Unfortunately, neither SpecFuzz nor SpecTaint evaluate memory consumption, so we cannot compare against them.

APPENDIX C
LARGE-SCALE EXPLOITABILITY EVALUATION

We evaluate the found gadgets across different metrics to provide a more detailed analysis of exploitability. Exploitability of a gadget depends on e.g., how close it is to the syscall entry to avoid unrelated noise or how long is the speculative window until the gadget is reached.

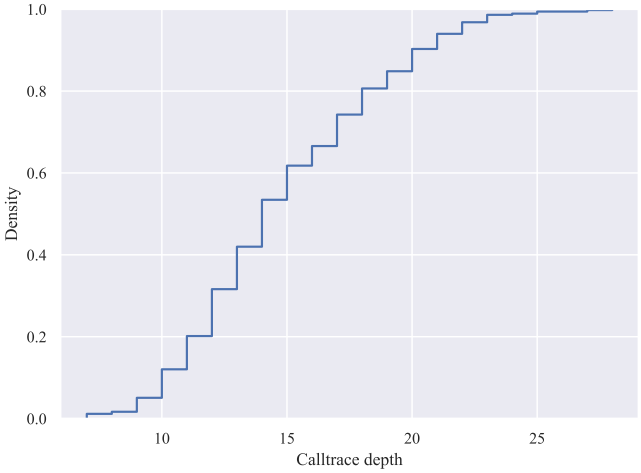


Fig. 8: Distribution of the smallest calltrace for the found gadgets.

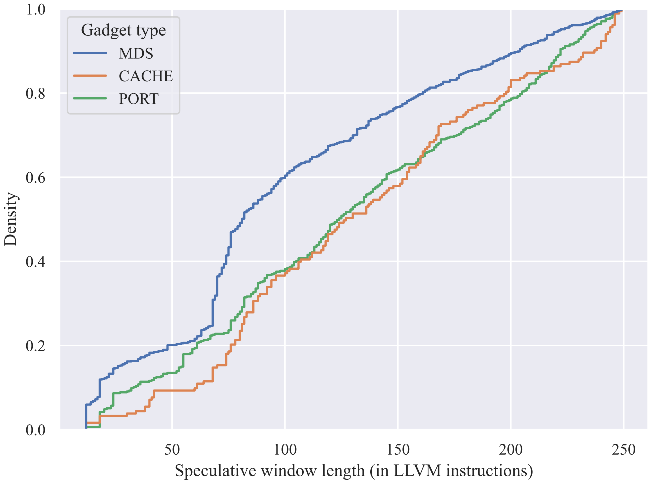


Fig. 9: Distribution of the smallest speculative length in LLVM instructions for the found gadgets.

Required call depth. We present the distance to the syscall entry by the depth of the calltrace in Figure 8. The first four to five functions called on a syscall entry are usually small and just direct execution towards the correct syscall handler, explaining the low number of unique gadgets found with a call depth below five. 50% of the gadgets are found with a call depth of a maximum of 9, suggesting that they are more likely to be exploitable by this metric. There was no significant difference in the call depth between the different gadget types.

Required speculation window length. In Figure 9, we present the length of the speculative window from the mispredicted branch until the leakage. More than 60% of MDS gadgets have a speculative window length of less than 100 LLVM instructions. Similarly, 50% of the CACHE and PORT gadgets have a speculative window length of less than 130 LLVM instructions. MDS gadgets, on average, require a smaller window since these gadgets leak through the access and do not require an

```
1  if ( arg == 0 || arg > MAX_NR_CONSOLES )
2     ret = -ENXIO;
3  else {
4     arg--;
5     currcons = arg;
6     console_lock();
7     WARN_CONSOLE_UNLOCKED();
8     if ( currcons >= MAX_NR_CONSOLES )
9         return -ENXIO;
10
11    if ( vc_cons[ currcons ].d )
12      ...
13 }
```

Listing 12: MDS-based gadget in the `vc_allocate` function used within the `vt_ioctl` function.

additional step to encode the secret in the cache. Such short speculative windows allow for easier exploitation and make the gadgets also applicable to processors with smaller speculative windows.

Appendix D
Additional Case Study

We present a case study found by Kasper that highlights the need for a dynamic analysis-based approach and the need for automated transient execution patch verification.

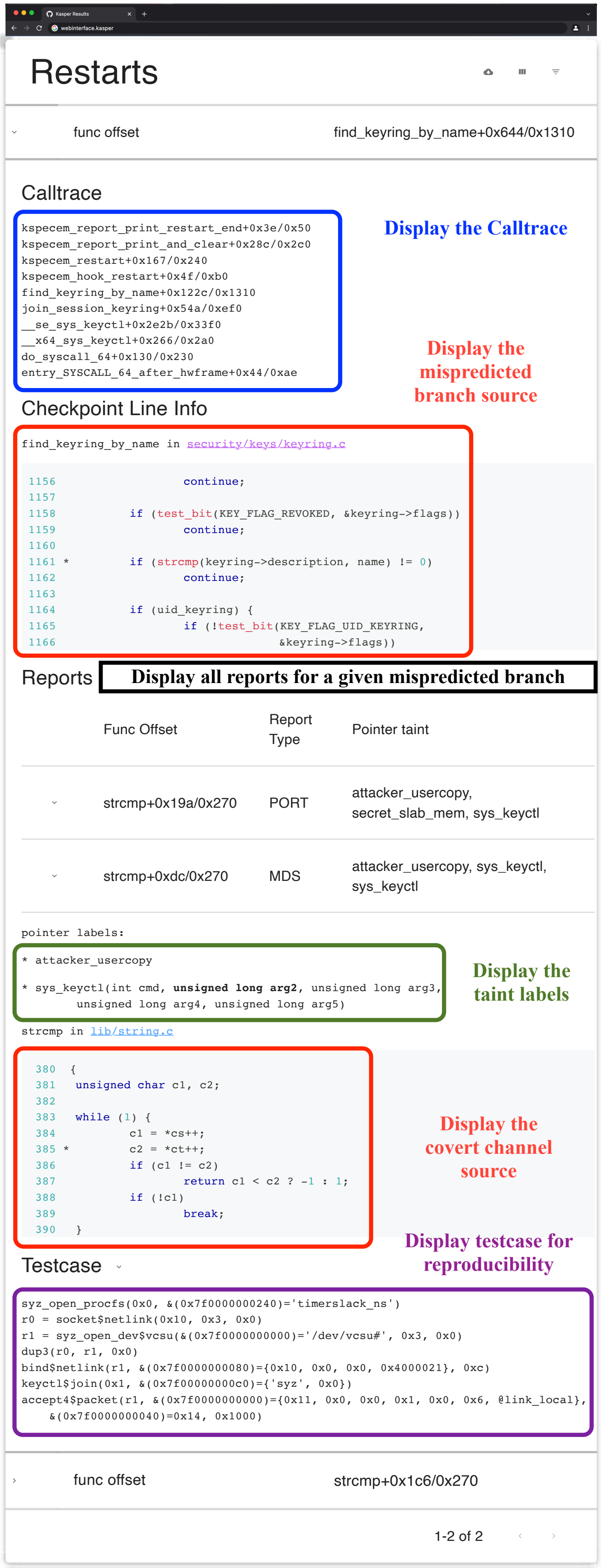
The gadget. Listing 12 shows an MDS-based gadget in the `vc_allocate` function, which is called by `vt_ioctl`. In this gadget, the attacker first supplies `arg` through an ioctl syscall argument, which Kasper marks as tainted. Then, Kasper emulates the mispredicted bounds check at line 1 and executes the *else* statement with an out-of-bounds value for `arg`. After a second mispredicted bounds check at line 8, Kasper detects an out-of-bounds memory access at line 11. Since `curcons` is a 32-bit value under full control of the attacker, this gadget allows an attacker to leak a large range of kernel memory. Note that a previous version of Kasper that used a basic implementation of nested speculation found this gadget, as it relies on two inverted branches (lines 1 and 8).

Drawback of static analysis. We identified an almost identical code snippet in `vc_setallocate`—which is very close to the other gadget, and also called from `vt_ioctl`—however, it was already (partially) mitigated through speculative array index masking. It is unclear why the kernel developers applied the mitigation to this gadget, but not to the other. At first glance, the only noticeable difference between the two gadgets is that this gadget receives user data from a `copy_from_user` call, whereas the other gadget receives user data from a syscall argument. However, upon closer inspection, it becomes clear why the kernel's static analysis tool [11] may not have identified it. The dataflow from the `copy_from_user` to the (partially) mitigated gadget is only separated by a *direct* call, whereas the dataflow from the syscall argument to the unmitigated gadget is separated by an *indirect* call. Since indirect calls are notoriously difficult to resolve statically [38], it is no surprise that the gadget was left unmitigated. This highlights the importance of a dynamic analysis-based approach, which can uncover gadgets beyond the reach of static analysis.

Drawback of manual mitigation verification. Upon further inspection of the mitigation in `vc_setallocate`, we found that it was applied incorrectly. Indeed, Kasper verifies that the mitigation is only partial. Namely, while the speculative array index masking ensures that the index becomes zero if it goes out-of-bounds, the decrement that follows (similar to line 4 in Listing 12) causes an integer underflow in transient execution. This demonstrates the importance of an automated tool such as Kasper, which can verify that manually-applied security patches work as intended.

Appendix E
Developer Interface

We have built a web interface to visualize all the necessary information retrieved from the database, so that kernel developers can easily analyze and patch the reported Kasper gadgets. Figure 10 presents a screenshot of the interface giving the kernel developer important information such as the call-trace, source code of the mispredicted branch and the leaking operation, taint information, and a testcase to reproduce the report.



Restarts

func offset find_keyring_by_name+0x644/0x1310

Calltrace

```
kspecem_report_print_restart_end+0x3e/0x50
kspecem_report_print_and_clear+0x28c/0x2c0
kspecem_restart+0x167/0x240
kspecem_hook_restart+0x4f/0xb0
find_keyring_by_name+0x122c/0x1310
join_session_keyring+0x54a/0xef0
__se_sys_keyctl+0x2e2b/0x33f0
__x64_sys_keyctl+0x266/0x2a0
do_syscall_64+0x130/0x230
entry_SYSCALL_64_after_hwframe+0x44/0xae
```

Display the Calltrace

Checkpoint Line Info

find_keyring_by_name in security/keys/keyring.c

```
                        continue;

                if (test_bit(KEY_FLAG_REVOKED, &keyring->flags))
                        continue;

                if (strcmp(keyring->description, name) != 0)
                        continue;

                if (uid_keyring) {
                        if (!test_bit(KEY_FLAG_UID_KEYRING,
                                      &keyring->flags))
```

Display the mispredicted branch source

Reports

Display all reports for a given mispredicted branch

	Func Offset	Report Type	Pointer taint
⌄	strcmp+0x19a/0x270	PORT	attacker_usercopy, secret_slab_mem, sys_keyctl
⌄	strcmp+0xdc/0x270	MDS	attacker_usercopy, sys_keyctl, sys_keyctl

```
pointer labels:
* attacker_usercopy

* sys_keyctl(int cmd, unsigned long arg2, unsigned long arg3,
        unsigned long arg4, unsigned long arg5)
```

Display the taint labels

strcmp in lib/string.c

```
{
        unsigned char c1, c2;

        while (1) {
                c1 = *cs++;
                c2 = *ct++;
                if (c1 != c2)
                        return c1 < c2 ? -1 : 1;
                if (!c1)
                        break;
        }
```

Display the covert channel source

Testcase ⌄

Display testcase for reproducibility

```
syz_open_procfs(0x0, &(0x7f0000000240)='timerslack_ns')
r0 = socket$netlink(0x10, 0x3, 0x0)
r1 = syz_open_dev$vcsu(&(0x7f0000000000)='/dev/vcsu#', 0x3, 0x0)
dup3(r0, r1, 0x0)
bind$netlink(r1, &(0x7f0000000080)={0x10, 0x0, 0x0, 0x4000021}, 0xc)
keyctl$join(0x1, &(0x7f00000000c0)={'syz', 0x0})
accept4$packet(r1, &(0x7f0000000000)={0x11, 0x0, 0x0, 0x1, 0x0, 0x6, @link_local},
    &(0x7f0000000040)=0x14, 0x1000)
```

› func offset strcmp+0x1c6/0x270

1-2 of 2 ‹ ›

Fig. 10: Screenshot of KASPER's web interface.