An Analysis of Speculative Type Confusion Vulnerabilities in the Wild

Ofek Kirzner Adam Morrison
Tel Aviv University

Abstract

Spectre v1 attacks, which exploit conditional branch misprediction, are often identified with attacks that bypass array bounds checking to leak data from a victim's memory. Generally, however, Spectre v1 attacks can exploit *any* conditional branch misprediction that makes the victim execute code incorrectly. In this paper, we investigate *speculative type confusion*, a Spectre v1 attack vector in which branch mispredictions make the victim execute with variables holding values of the wrong type and thereby leak memory content.

We observe that speculative type confusion can be inadvertently introduced by a compiler, making it extremely hard for programmers to reason about security and manually apply Spectre mitigations. We thus set out to determine the extent to which speculative type confusion affects the Linux kernel. Our analysis finds exploitable and potentially-exploitable *arbitrary* memory disclosure vulnerabilities. We also find many *latent* vulnerabilities, which could become exploitable due to innocuous system changes, such as coding style changes.

Our results suggest that Spectre mitigations which rely on statically/manually identifying "bad" code patterns need to be rethought, and more comprehensive mitigations are needed.

1 Introduction

Spectre attacks [13, 30, 36, 43, 45, 46, 54] exploit processor control- or data-flow speculation to leak data from a victim program's memory over a microarchitectural covert channel. A Spectre attack maneuvers the processor to mispredict the correct execution path of an instruction sequence in the victim, referred to as a *Spectre gadget*. The gadget's misspeculated execution acts as a "confused deputy" and accesses data from an attacker-determined location in the victim's address space. Although the mispredicted instructions are *transient* [18,45]—the CPU eventually discards them without committing their results to architectural state—data-dependent traces of their execution remain observable in microarchitectural state, such as the cache. These data-dependent side effects form a covert channel that leaks the accessed data to the attacker.

Spectre attacks pose a serious threat to monolithic operating system (OS) kernels. While Spectre attacks can only leak data architecturally accessible to the victim, a victim that is an OS kernel typically has all physical memory mapped in its virtual address space and thus architecturally accessible [5, 23]. Moreover, kernels expose a large attack surface (e.g., hundreds of system calls) through which an attacker can trigger Spectre gadgets.

```
if (x < array1_len) { // branch mispredict: taken
  y = array1[x];      // read out of bounds
  z = array2[y * 4096]; } // leak y over cache channel
```

(a) Bounds check bypass.

```
void syscall_helper(cmd_t* cmd, char* ptr, long x) {
  // ptr argument held in x86 register %rsi
  cmd_t c = *cmd;   // cache miss
  if (c == CMD_A) { // branch mispredict: taken
    ... code during which x moves to %rsi ...
  }
  if (c == CMD_B) { // branch mispredict: taken
    y = *ptr;  // read from addr x (now in %rsi)
    z = array[y * 4096]; // leak y over cache channel
  }
  ... rest of function ...
```

(b) Type confusion.

Listing 1: Spectre gadgets for exploiting conditional branch prediction. Data in **red boxes** is attacker-controlled.

Since speculative execution is fundamental to modern processor design, processor vendors do not plan to mitigate Spectre attacks completely in hardware [8, 37, 39].[1] Vendors instead suggest using software mitigations to restrict speculation [8, 37]. To minimize their performance impact, most software mitigations target specific "templates" of potentially vulnerable gadgets, which are identified with static or manual analysis [38, 51, 57, 88].[2]

In this paper, we focus on conditional branch prediction

[1]In contrast, Meltdown-type attacks [52,70,82,83,85] exploit an Intel processor implementation artifact (addressed in future processors [39]), wherein a load targeting *architecturally inaccessible* data (e.g., in another address space) can execute before being discarded due to a virtual memory exception.

[2]We discuss more comprehensive software mitigations—which, unfortunately, have high performance overheads—in § 7.

Spectre attacks (so-called "variant 1" [45]). These attacks are often characterized as *bounds check bypass* attacks, which exploit misprediction of an array bounds check to perform an out-of-bounds access and leak its result (Listing 1a). Deployed software mitigations in compilers and OS kernels target this type of gadget template [38, 51, 57].

Generally, however, a Spectre attack is defined as exploiting conditional branch prediction to make the processor *"temporarily violate program semantics by executing code that would not have been executed otherwise"* [45]—and a bounds check bypass is just one example of such a violation. Speculative *type confusion* is a different violation, in which misspeculation makes the victim execute with some variables holding values of the wrong type, and thereby leak memory content.

Listing 1b shows an example *compiler-introduced* speculative type confusion gadget, which causes the victim to dereference an attacker-controlled value. In this example, the compiler emits code for the first `if` block that clobbers the register holding a (trusted) pointer with an untrusted value, based on the reasoning that if the first `if` block executes, then the second `if` block will not execute. Thus, if the branches mispredict such that both blocks execute, the code in the second `if` block leaks the contents of the attacker-determined location. In contrast to the bounds check bypass attack, here the attacker-controlled address has no data-dependency on the branch predicate, nor does the predicate depend on untrusted data. Consequently, *this gadget would not be protected by existing OS kernel Spectre mitigations, nor would programmers expect it to require Spectre protection.*

To date, speculative type confusion has mainly been hypothesized about, and even then, only in the context of object-oriented polymorphic code [18] or as a vector for bypassing bounds checks [32, 56]. Our key driving observation in this paper is that speculative type confusion may be much more prevalent—as evidenced, for instance, by Listing 1b, which does not involve polymorphism or bounds checking. Accordingly, we set out to answer the following question: *are OS kernels vulnerable to speculative type confusion attacks?*

1.1 Overview & contributions

We study Linux, which dominates the OS market share for server and mobile computers [90]. In a nutshell, not only do we find exploitable speculative type confusion vulnerabilities, but—perhaps more disturbingly—our analysis indicates that OS kernel security currently rests on shaky foundations. There are many latent vulnerabilities that are not exploitable only due to serendipitous circumstances, and may be rendered exploitable by different compiler versions, innocuous code changes, deeper-speculating future processors, and so on.[3]

[3]Indeed, we make no security claims for these "near miss" vulnerabilities; some of them may be exploitable in kernel versions or platforms that our analysis—which is not exhaustive—does not cover.

Attacker-introduced vulnerabilities (§ 4) Linux supports untrusted user-defined kernel extensions, which are loaded in the form of eBPF[4] bytecode programs. The kernel verifies the safety of extensions using static analysis and compiles them to native code that runs in privileged context. The eBPF verifier does not reason about speculative control flow, and thus successfully verifies eBPF programs with speculative type confusion gadgets. eBPF emits Spectre mitigations into the compiled code, but these only target bounds check bypass gadgets. Consequently, we demonstrate that an unprivileged user can exploit eBPF to create a Spectre *universal read gadget* [55] and read arbitrary physical memory contents at a rate of 6.7 KB/sec with 99% accuracy.

Compiler-introduced vulnerabilities (§ 5) We show that C compilers can emit speculative type confusion gadgets. While the gadgets are blocked by full Spectre compiler mitigation modes (e.g., speculative load hardening (SLH) [21]), these modes have high performance overheads (§ 7), and in GCC must be manually enabled per-variable. Optional low-overhead mitigation modes in Microsoft and Intel compilers do not block these gadgets. Motivated by these findings, we perform a binary-level analysis of Linux to determine whether it contains speculative type confusion introduced by compiler optimizations. We find several such cases. In assessing potential exploitability of these cases, we investigate how x86 processors resolve mispredicted branches. We find that Spectre gadgets which today may be considered unexploitable are actually exploitable, which may be of independent interest.

Polymorphism-related vulnerabilities (§ 6) The Linux kernel makes heavy use of object-oriented techniques, such as data inheritance and polymorphism, for implementing kernel subsystem interfaces. The related indirect function calls are protected with retpolines [81], which essentially disable indirect call prediction. To claw back the resulting lost performance, Linux replaces certain indirect calls with direct calls to one of a set of legal call targets, where the correct target is chosen using conditional branches [9, 24]. Unfortunately, this approach opens the door to speculative type confusion among the different targets implementing a kernel interface. We perform a source-level analysis on Linux to find such vulnerabilities. We identify dozens of *latent* vulnerabilities, namely: vulnerable gadgets which are not exploitable by chance, and could become exploitable by accident. For example, we find gadgets in which the attacker controls a 32-bit value, which cannot represent a kernel pointer on 64-bit machines. But if a future kernel version makes some variables 64-bit wide, such gadgets would become exploitable.

1.2 Implications

Our work shows that speculative type confusion vulnerabilities are more insidious than speculative bounds check bypasses, with exploitable and latent vulnerabilities existing in

[4]Extended Berkeley Packet Filter [11, 68].

kernel code. Given the existence of compiler-introduced vulnerabilities, we question the feasibility of the current Linux and GCC mitigation approach, which relies on developers manually protecting "sensitive" variables [51], likely due to equating Spectre v1 with bounds check bypasses. While comprehensive mitigations, such as SLH [21], can block all Spectre attacks, they impose significant overhead on kernel operations (up to 2.7×, see § 7). It is also unclear whether static analysis [22, 31] can be incorporated into the OS kernel development process to guarantee absence of speculative type confusion vulnerabilities. In short, current Spectre mitigations in OS kernels require rethinking and further research.

2 Background

2.1 Out-of-order & speculative execution

Modern processors derive most of their performance from two underlying mechanisms: out-of-order (OoO) execution [79] and speculation [34].

OoO execution A processor core consists of a *frontend*, which fetches instruction from memory, and a *backend*, responsible for instruction execution. A fetched instruction is *decoded* into internal micro-operations (μ-ops), which are then *dispatched* to a *reservation station* and await execution. Once the operands of a μ-op become available (i.e., have been computed), it is *issued* to an execution unit where it is *executed*, making its result available to dependent μ-ops. Hence, μ-ops may execute out of program order. To maintain the program order and handle exceptions, μ-ops are queued into a reorder buffer (ROB) in program order. Once a μ-op reaches the ROB head and has been executed, it gets *retired*: its results are committed to architectural state and any pipeline resources allocated to it are freed.

Speculative execution To execute instructions as soon as possible, the processor attempts to *predict* the results of certain (usually long latency) μ-ops. The prediction is made available to dependent μ-ops, allowing them to execute. Once the predicted μ-op executes, the backend checks if the μ-op's output was correctly predicted. If so, the μ-op proceeds to retirement; otherwise, the backend *squashes* all μ-ops following the mispredicted μ-op in the ROB and reverts its state to the last known correct state (which was checkpointed when the prediction was made). We refer to the maximum amount of work that can be performed in the shadow of a speculative μ-op as the *speculation window*. It is determined by the latency of computing the predicted μ-op's results and the available microarchitectural resources (e.g., the size of the ROB limits how many μ-ops can be in flight). We consider *control-flow speculation*, described in the following section.

2.2 Branch prediction

To maximize instruction throughput, the processor performs branch prediction in the frontend. When a branch is fetched, a *branch predictor unit* (BPU) predicts its outcome, so that the frontend can continue fetching instructions from the (predicted) execution path without stalling. The branch is *resolved* when it gets executed and the prediction is verified, possibly resulting in a squash and re-steering of the frontend. Notice that *every* branch is predicted, even if its operands are readily available (e.g., in architectural state), because figuring out availability of operands is only done in the backend.

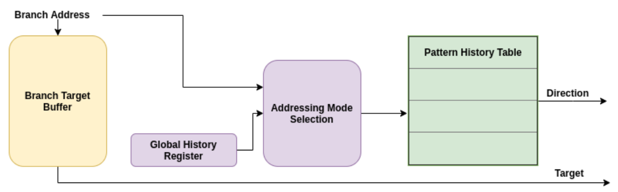


Figure 1: BPU Scheme

We assume the branch predictor unit design shown in Figure 1, which appears to match Intel's BPU [26, 97]. The BPU has two main components: an outcome predictor, predicting the direction of conditional branches, and a *branch target buffer* (BTB), predicting the target address of indirect branches. The outcome predictor stores 2-bit saturating counters in a *pattern history table* (PHT). A branch's outcome is predicted based on a PHT entry selected by hashing its program counter (PC). The PHT entry is selected in one of two addressing modes, depending on the prediction success rate: 1- or 2-level prediction. The 1-level predictor uses only the PC, whereas the 2-level predictor additionally hashes a *global history register* (GHR) that records the outcome of the previously encountered branches.

2.3 Cache covert channels

To hide memory latency, the processor contains fast memory buffers called *caches*, which hold recently and frequently accessed data. Modern caches are *set-associative*: the cache is organized into multiple *sets*, each of which can store a number of cache lines. The cache looks up an address by hashing it to obtain a set, and then searching all cache lines in that set.

Changes in cache state can be used to construct a covert channel. Consider transmission of a B-bit symbol x. In a FLUSH+RELOAD channel [92], (1) the receiver flushes 2^B lines from the cache; (2) the sender accesses the x-th line, bringing it back into the cache; and (3) the receiver measures the time it takes to access each line, identifying x as the only cache hit. FLUSH+RELOAD requires the sender and receiver to share memory. A PRIME+PROBE channel avoids this requirement by using evictions instead of line fills [53].

2.4 Transient execution attacks

Transient execution attacks overcome architectural consistency by using the microarchitectural traces left by *transient*—doomed to squash—μ-ops to leak architecturally-inaccessible data to the attacker. Meltdown-type attacks [18, 52, 70, 82, 85] extract data across architectural protection domains by exploiting transient execution after a hardware exception. Our focus, however, is on Spectre-type attacks [13, 30, 36, 43, 45, 46, 54], which exploit misprediction. Spectre-type attacks maneuver the victim into leaking its own data, and are limited to the depth of the speculation window. Spectre v1 (Spectre-PHT [18]) exploits misprediction of conditional branch outcomes. Spectre v2 (Spectre-BTB [18]) exploit misprediction of indirect branch target addresses. The original Spectre v2 attacks poisoned the BTB to redirect control-flow to arbitrary locations; but as we shall see, even mispredicting a legal target is dangerous (§ 6). Other variants target return address speculation [46, 54] or data speculation [36].

3 Threat model

We consider an attacker who is an unprivileged (non-root) user on a multi-core machine running the latest Linux kernel. The attacker's goal is to obtain information located in the kernel address space, which is not otherwise accessible to it.

We assume the system has all state-of-the-art mitigations against transient execution attacks enabled. In particular, the attacker cannot mount cross-protection domain Meltdown-type attacks to directly read from the kernel address space.

We assume that the attacker knows kernel virtual addresses. This knowledge can be obtained by breaking kernel address space layout randomization (KASLR) using orthogonal side-channel attacks [19, 84, 86] or even simply from accidental information leak bugs in the kernel.

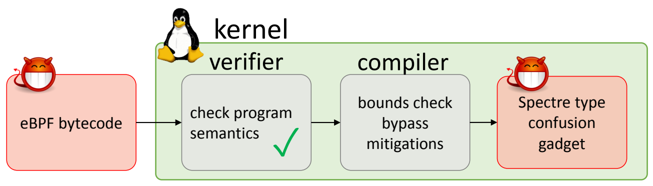
4 Speculative type confusion in eBPF

Linux's extended Berkeley Packet Filter (eBPF) [68] subsystem strives to let the Linux kernel safely execute untrusted, user-supplied kernel extensions in privileged context. An eBPF extension is "attached" to designated kernel events, such as system call execution or packet processing, and is executed when these events occur. eBPF thereby enables applications to customize the kernel for performance monitoring [3], packet processing [2], security sandboxing [4], etc.

Unprivileged users can load eBPF extensions into the kernel as of Linux v4.4 (2015) [75].[5] An eBPF extension is loaded in the form of a bytecode program for an in-kernel virtual machine (VM), which is limited in how it can interact with the rest of the kernel. The kernel statically verifies the safety of loaded eBPF programs. On the x86-64 and Arm architectures, the kernel compiles eBPF programs to native code; on other architectures, they run interpreted.

Both eBPF verification and compilation do not consider speculative type confusion, which allows an attacker to load eBPF programs containing Spectre gadgets and thus read from anywhere in the kernel address space (Figure 2). In the following, we describe the eBPF security model (§ 4.1), detail its vulnerability (§ 4.2), and describe our proof-of-concept attack (§ 4.3).

[5] An `unprivileged_bpf_disable` configuration knob exists for disallowing unprivileged access to eBPF; it is not set by default.


kernel
verifier
compiler
eBPF bytecode
check program semantics
bounds check bypass mitigations
Spectre type confusion gadget
Figure 2: eBPF speculative type confusion attack.

4.1 eBPF security model

In general, eBPF does not enforce safety at run time, but by statically verifying that the eBPF program maintains memory safety and is otherwise well-behaved. (One exception to this principle are Spectre mitigations, discussed below.) An eBPF program can only call one of a fixed set of *helper* functions in the kernel. The allowed set of helpers depends on the type of eBPF program and on the privileges of the user that loaded it.

An eBPF program is restricted to accessing a fixed set of memory regions, known at compile time, including a fixed-size stack and a *context*, which is a program-specific object type that stores the program's arguments. An eBPF program can further access statically-sized key/value dictionaries called *maps*. The size of a map's keys and values is fixed at map creation time. A map's data representation (array, hash table, etc.) is similarly specified on creation. Maps can be shared by different eBPF programs and can also be accessed by user processes. An eBPF program that successfully looks up a value in a map receives a pointer to the value, and so can manipulate it in-place.

eBPF verification The kernel statically verifies the safety of an eBPF program in two steps. The first step performs control-flow validation, to verify that the program (which runs in privileged context) is guaranteed to terminate in a fixed amount of time. This property is verified by checking that the program does not contain loops and does not exceed a certain fixed size. The second step verifies the program's memory safety. Memory safety is verified by enumerating every possible execution flow to prove that every memory access is safe. When processing a flow, the verifier maintains a *type* for each register and stack slot, and checks that memory accesses are valid with respect to these types.

The verifier tracks whether each register is uninitialized, holds a scalar (non-pointer) value, or holds a pointer. Pointers are further typed according to the region they point to: the stack, the context, a map, a value from a map, and so on. The verifier also tracks whether a pointer is non-NULL and maintains bounds for the pointer's offset from its base object. Using this information, the verifier checks various safety properties, such as:

- For every memory access, the operand register *R* contains a pointer type, *R* ≠ NULL, and *R* points to within its base object.
- If a stack slot is read, then the program has previously written to it. (This check prevents leaking of kernel data that was previously stored in that location.)
- Pointers are not cast to scalars. (This check prevents kernel addresses from leaking to userspace, since such scalars can be stored to a map and read by the user.)

eBPF also allows certain program types to access kernel data structures such as socket and packet objects, whose size may not be known at compile time. In such cases, the runtime stores the object size in the program's context, and the verifier checks that any pointer to these objects undergoes bounds checking before being dereferenced. We do not discuss this further, since our attack does not exploit this functionality.

Compile-time Spectre mitigation Following the disclosure of Spectre, the eBPF verifier was extended to patch eBPF programs with run-time mitigations for Spectre bounds check bypass vulnerabilities [15,76]. The verifier rewrites any array-map access and any pointer arithmetic operation so that they are guaranteed to be within the base object's bounds. For example, `A[i]` is rewritten as `A[i & (A.size-1)]`, where `A.size` is rounded up to be a power of 2. Pointer arithmetic is handled similarly, leveraging the verifier's knowledge about base objects' size and pointer offsets.

4.2 Verifier vulnerability

When verifying memory safety, the eBPF verifier enumerates only correct execution paths (i.e., that comply with the semantics of the architecture's instruction set). The verifier does not reason about the semantically incorrect execution paths that arise due to (mis)speculative execution. As a result, the verifier can conclude that a memory read is safe, but there may be a misspeculated execution flow in which that instruction is unsafe. Listing 2a shows an example.

In this example, the semantics of the program are such that the execution of lines 3 and 5 is mutually exclusive: line 3 executes if and only if `r0`=`0` and line 5 executes if and only if `r0`=`1`. Therefore, the verifier reasons that the only flow in which line 5 executes and memory is read is when `r6` points to a stack slot, and accepts the program. Specifically, when the verifier's analysis reaches line 2, it enumerates two cases:

- If the condition on line 2 evaluates to TRUE, `r6`'s type remains a valid stack variable.
- If the condition on line 2 evaluates to FALSE, `r6`'s type changes to scalar, but the verifier learns that `r0` is 0. Therefore, when it subsequently reaches line 4, it reasons that the condition there must evaluate to TRUE, and does not consider line 5 in these flows.

In both cases, every execution flow the verifier considers is safe. Moreover, the load in line 5 is not rewritten with Spectre mitigation code, because the pointer it dereferences is verified to point to the stack (and thus within bounds). Nevertheless, if an attacker manages to (1) make the dereference of `r0` (line 1) be a cache miss, so that the branches take a long time to resolve; and (2) mistrain the branch predictor so that both branches predict "not taken" (i.e., do not execute the `goto`), then the resulting transient execution sets `r6` to the attacker-controlled value in `r9` (line 3), dereferences that value (line 5), and leaks it (line 6).

```
// r0 = ptr to a map array entry (verified ≠ NULL)
// r6 = ptr to stack slot (verified ≠ NULL)
// r9 = scalar value controlled by attacker
```

```
1    r0 = *(u64 *)(r0) // miss
2  A:if r0 != 0x0 goto B
3    r6 = r9
4  B:if r0 != 0x1 goto D
5    r9 = *(u8 *)(r6)
6  C:r1 = M[(r9&1)*512];//leak
7  D:...
```

(a) Passes verification.

```
1    r0 = *(u64 *)(r0) // miss
2  A:if r0 == 0x0 goto B
3    r6 = r9
4  B:if r0 != 0x0 goto D
5    r9 = *(u8 *)(r6)
6  C:r1 = M[(r9&1)*512];//leak
7  D:...
```

(b) Fails verification.

Listing 2: eBPF verification vulnerability: leaking a bit (eBPF bytecode; *rX* stand for eBPF bytecode registers).

We remark that in practice, the verifier does not maintain perfect information about each register, and so may end up enumerating some impossible execution flows. (I.e., the verifier enumerates an over-approximation of the correct execution flows.) Consequently, the verifier inadvertently rejects some speculative type confusion eBPF gadgets. Listing 2b shows an example. For scalar values, the verifier maintains either a register's exact value or a possible range of values. When the verifier considers the case of the condition on line 2 evaluating to FALSE, it cannot track the implication that `r0`≠`0`, as that cannot be represented with a single range. Since the verifier has no additional information about `r0`, it goes on to consider a continuation of the flow in which the condition in line 4 also evaluates to FALSE. In this flow, line 5 dereferences a scalar value, and so the program is rejected. This rejection is accidental. The goal of eBPF developers is to increase the precisions of the verifier's information tracking, and so under current development trends, we would expect the verifier to eventually accept Listing 2b. Improving the verifier's precision is not a problem in and of itself, if eBPF adopts general Spectre mitigations (i.e., not only bounds enforcing).

4.3 Proof of concept exploit

We now describe and evaluate a proof of concept exploit for the eBPF vulnerability. The exploit implements a universal read gadget [55], which allows the attacker to read from any kernel virtual address. This ability allows the attacker to read all of the machine's physical memory, as the kernel maps all available physical memory in its virtual address space [5].

The eBPF bytecode is designed to easily map to the x86 architecture, which the eBPF compiler leverages to map eBPF registers to x86 registers and eBPF branches to x86 branches. To guarantee that our victim eBPF program has the required structure (Listing 2a), we manually encode its bytecode instead of using a C-to-eBPF compiler. The kernel then translates the bytecode to native x86 code in the obvious way.

The main challenge faced by the exploit is how to mistrain the branch predictor, so that two conditional branches whose "not taken" conditions are mutually exclusive *both* get predicted as "not taken." Two further challenges are (1) how to evict the value checked by these branches from the cache, so that their resolution is delayed enough that the misspeculated gadget can read and leak data; and (2) how to observe the leaked data. These cache-related interactions are non-trivial to perform because the eBPF program runs in the kernel address space and the attacker, running in a user process, cannot share memory with the eBPF program. (The main method of interaction is via eBPF maps, but processes can only manipulate maps using system calls, not directly.)

Mistraining the branch predictor A common branch mistraining technique in Spectre attacks is to repeatedly invoke the victim with valid input (e.g., an in-bounds array index) to train the branch predictor, and then perform the attack with an invalid input (e.g., an out-of-bounds array index), at which point the relevant branch gets mispredicted. This technique does not apply in our case: we need to train two branches whose "not taken" conditions are mutually exclusive to both get predicted as "not taken." This means that no matter what input the eBPF victim gadget is given, at least one of the branches is always taken in its correct execution. In other words, we cannot get the branch predictor into a state that is *inconsistent* with a correct execution by giving it only examples of correct executions.

To address this problem, we use *cross address-space out-of-place* branch mistraining [18]. Namely, we set up a "shadow" of the natively-compiled eBPF program in the attacker's process, so that the PHT entries (§ 2.2) used to predict the shadow branches also get used to predict the victim branches in the kernel.[6] In our shadow, however, the branches are set up so the "not taken" conditions are not mutually exclusive, and so can be trained to both predict "not taken" (Listing 3).

To ensure PHT entry collision between the shadow and victim branches, we must control the following factors, upon which PHT indexing is based: (1) the state of the GHR and (2) the BPU-indexing bits in the branches' virtual addresses.

To control the GHR, we prepend a "branch slide" (Listing 4) to both the victim and its shadow. (For the eBPF victim, we generate the branch slide using appropriate eBPF bytecode, which the kernel compiles into the native code shown

[6]Note that this approach depends on the branch predictor state not being cleared when the processor switches to privileged mode on kernel entry. This is indeed the case in current processors.

```
// addresses A' and B' collide in the PHT
// with addresses A and B in Listing 2a
A': if r0 == 0x0 goto B'
    // dummy register assignment
B': if r0 == 0x0 goto C'
    // dummy pointer dereference
C': ...
```

Listing 3: Mistraining the branch predictor.

```
         mov    $0x1,%edi
         cmp    $0x0,%rdi
         jne    L1
L1:      cmp    $0x0,%rdi
         jne    L2
         ...
L(n-1):  cmp    $0x0,%rdi
         jne    Ln
Ln:      # exploit starts here
```

Listing 4: Branch slide

in the listing.) Execution of the branch slide puts the GHR into the same state both when the shadow is trained and when the victim executes.

To control address-based indexing, we need the address of the shadow gadget branches to map to the same PHT entries as the victim eBPF program's branches. We cannot create such a PHT collision directly, by controlling the shadow gadget's address, since we do not know the victim's address or the hash function used by the branch predictor. We can, however, perform a "brute force" search to find a collision, as prior work has shown that the PHT on Intel processors has 2^{14} entries [26]. We describe our collision search algorithm later, since it relies on our mechanism for leaking data.

Cache line flushing We need the ability to flush a memory location out of the victim's cache, for two reasons. First, we need to cause the read of the value checked by the branches to miss in the cache, so that the resulting speculation window is large enough to read and leak the secret data. Second, one of the simplest ways of leaking the data is via a FLUSH+RELOAD cache covert channel [92], wherein the transient execution brings a previously flushed secret-dependent line into the cache. Line 6 in Listing 2a shows an example, which leaks over an eBPF array map, `M`. Notice that we mask the secret value to obtain a valid offset into the array, to satisfy the eBPF verifier. As a result, this example leaks a single bit.

Unfortunately, eBPF programs cannot issue a `clflush` instruction to flush a cache line. This problem can be sidestepped in a number of ways. We use a clever technique due to Horn [35]. The basic idea is to perform the required cache line flushes by having *another* eBPF program, running on a different core, write to these cache lines, which reside in a shared eBPF array map. These writes invalidate the relevant cache lines in the victim's cache, resulting in a cache miss the next time the victim accesses the lines. After mounting the attack, the attacker runs a third eBPF program on the victim's core to perform the timing measurements that deduce which line the transient execution accessed, and thereby the secret:

```
r0 = CALL ktime_get_ns()
r1 = M[b]  // b is 0*512 or 1*512
r2 = CALL ktime_get_ns()
return r2 - r0 // if small -> secret is b
```

This approach leverages the fact that eBPF programs can perform fine-grained timing measurements by invoking the `ktime_get_ns()` kernel helper. This is not fundamental for the attack's success, however. Similarly to what has been shown for JavaScript [71], we could implement a fine-grained "clock" by invoking eBPF code on another core to continuously increment a counter located in a shared eBPF map.

Finding address-based PHT collisions To place our shadow branches into addresses that get mapped to the same PHT entries as the victim's branches (whose address is unknown), we perform the following search algorithm.

We allocate a 2 MB buffer and then, for each byte in the buffer, we copy the shadow gadget to that location and check for a collision by trying the attack. We first mistrain the branch predictor by repeatedly executing the shadow gadget (whose branches' PHT entries are hoped to collide with the victim's). We then invoke the in-kernel victim gadget, configured (by setting the array entry read into `r0`) so that its correct execution does not leak (i.e., does not execute line 6 in Listing 2a). If no leak occurs—i.e., both timing measurements of `M[b]` indicate a cache miss—we do not have a collision. If a leak occurs, we may still not have a collision: the victim may have leaked its own stack variable by executing line 5, either due to the initial BPU state or if we only have a PHT collision with the second branch. To rule these possibilities out, we try the attack again, this time with the relevant bit flipped in that stack variable (which is done by invoking the victim with a different argument). If the leaked bit flips too, then we do not have a collision; otherwise, we do. Once found, a collision can be reused to repeat attacks against the victim. If the search fails, the attacker can re-load the victim and retry the search.

4.3.1 Evaluation

We use a quad-core Intel i7-8650U (Kaby Lake) CPU. The system runs Ubuntu 18.04.4 LTS with Linux 5.4.11 in a workstation configuration, with applications such as Chrome, TeXstudio, and Spotify running concurrently to the experiments.

PHT collisions We perform 50 experiments, each of which searches for a shadow gadget location that results in PHT collisions with a freshly loaded victim. Successful searches take 9.5 minutes on average and occur with 92% probability

found collision?	average	min.	max.	median
success (46/50)	9.5 min.	20 sec.	45 min.	8.5 min.
failure (4/50)		≈ 53 min		

Table 1: Times to find PHT collision with victim (50 experiments).

retries	success rate	transmission rate
1	99.9%	55,416 bps
2	98.7%	28,712 bps
10	100%	5,881 bps
100	100%	584 bps

Table 2: Accuracy and capacity of the eBPF covert channel.

(Table 1). Our search algorithm can be optimized, e.g., by considering only certain addresses (related to BPU properties and/or kernel buffer alignment). The search, however, is not a bottleneck for an attack, since once a location for the shadow gadget is found, it can be reused for multiple leaks. We therefore do not invest in optimizing the search step.

Covert channel quality We attempt to leak the contents of one page (4096 bytes) of kernel memory, which is pre-filled with random bytes. We leak this page one bit at a time, as described above. The only difference from Listing 2a is that our victim eBPF program receives an argument specifying which bit to leak in the read value, instead of always leaking the least significant bit. To leak a bit, we *retry* the attack k times, and output the majority value leaked over the k retries.

Table 2 shows the resulting accuracy (percentage of bits leaked correctly) and throughput (bits/second) of the overall attack, as a function of the number of retries. Since all steps are carried out using the same shadow gadget location, we do not account for the initial search time. The attack reads from arbitrary memory locations at a rate of 6.7 KB/sec with 99% accuracy, and 735 bytes/sec with 100% accuracy. (The success rate with 2 retries is lower due to an implementation artifact in our majority-taking computation.)

5 Compiler-introduced speculative type confusion

In principle, compiler optimizations can create speculative type confusion gadgets in the emitted code (Listing 1b shows a hypothetical example). Here, we first show that this is not a theoretical concern: deployed compilers can generate speculative type confusion gadgets, and certain Spectre compiler mitigations do not identify or block such gadgets (§ 5.1). Motivated by this finding, we perform a binary analysis on Linux and find that it contains potential compiler-introduced vulnerabilities (§ 5.2).

5.1 Compilers emit gadgets

We test different versions of several compilers: GCC, Clang, Intel ICC (from Intel Parallel Studio), and Microsoft Visual

Studio (MSVC). We find that all of them can compile C code into x86 code that contains a speculative type confusion gadget. Table 3 summarizes the results.

Listing 5 shows an example for GCC; the other compilers produce similar results for similar code. Here, the code in the first `if` block overwrites the `rdi` register (argument `p`) with the `rsi` register (attacker-controlled argument `x`). The compiler performs this overwrite because it enables using the same instruction for the assignment to `foo` at the end of the function. The compiler also reasons that the write to `*q` might modify `predicate` (if `q` points to `predicate`), and thus `predicate` should be re-checked after the first `if` block. The compiler's analysis does not understand that in a correct execution, the first `if` block executing implies that the second `if` block does not execute, even if `q` points to `predicate`. However, if the attacker mistrains the branches such that both predict "not taken," the resulting transient execution dereferences the attacker-controlled value `x` and leaks its value. Using the mistraining technique of § 4.3, we verify that this is possible.

Spectre mitigations efficacy We test whether each compiler's Spectre mitigations apply protection in our example.

Clang/LLVM: Implements a generic mitigation called speculative load hardening (SLH) [21]. SLH inserts branchless code that creates a data dependency between each load's address and *all* prior conditional branch predicates. SLH thus successfully protects the gadget in our example, but at a high performance cost (§ 7).

MSVC: Supports several mitigation levels. The most aggressive mitigation (`/Qspectre-load`) inserts an `lfence` speculation barrier after every load instruction. This mitigation applies in our example. However, its documentation warns that "the performance impact is high" [59]. In contrast, MSVC's recommended Spectre v1 mitigation, `/Qspectre` [58], targets bounds check bypass attacks and does not insert any speculation barriers in our example.

ICC: Similarly to MSVC, ICC supports several mitigation levels [38]. It offers two full mitigation options, based on speculation barriers (`all-fix`) or SLH (`all-fix-cmov`), with the former documented as having "the most run-time performance cost" [38]. Both options apply in our example. ICC also offers a "vulnerable code pattern" mitigation, which does not insert speculation barriers in our example.

GCC: Does not support a whole-program mitigation. It offers a compiler intrinsic for safely accessing values in the face of possible misspeculation. However, programmer who equate Spectre v1 with bounds check bypass have no reason to use this intrinsic in our example, so we consider GCC's mitigation inapplicable in our case.

5.2 Finding compiler-introduced gadgets

To find potential compiler-introduced speculative type confusion vulnerabilities in the wild, we perform a binary-level static analysis of Linux 5.4.11, compiled with different GCC versions and optimization flags.

compiler	emits gadget?	mitigates gadget?
Clang/LLVM (v3.5, v6, v7.01, v10.0.1)	yes	yes
MSVC (v16)	yes	suggested: no; full: yes
ICC (v19.1.1.217)	yes	lightweight: no; full: yes
GCC (v4.8.2, v7.5.0)	yes	N/A

Table 3: Compilers introducing speculative type confusion.

```
volatile char A[256*512];
bool predicate;
char* foo;

void victim(char *p,
            uint64_t x,
            char *q) {
  unsigned char v;

  if (predicate) {
    p = (char *) x;
    *q |= 1;
  }
  if (!predicate) {
    v = A[(*p) * 512];
  }
  foo = p;
}
```

(a) C code

```
# args: p in %rdi
#       x in %rsi
#       q in %rdx

  # first "if":
  cmpb   $0x0,(predicate)
B1:je    L1 # skip 2nd if
  # assignment to p:
  mov    %rsi,%rdi
  # assignment to q:
  orb    $0x1,(%rdx)
  # second "if":
  cmpb   $0x0,(predicate)
B2:jne   L2
  # deref p & leak
L1:movsbl (%rdi),%eax
  shl    $0x9,%eax
  cltq
  movzbl A(%rax),%eax
L2:mov   %rdi,(foo)
  retq
```

(b) Emitted x86 code.

Listing 5: Example of C code compiled into a speculative type confusion gadget (GCC 4.8.2, -O1). Argument x is attacker-controlled.

Goal & methodology We set to find out if the kernel can be maneuvered (via transient execution) to dereference a user-supplied address, which is the core of the attack. We explicitly do not consider if or how the result of the dereference can be leaked, for the following reasons. Once a secret enters the pipeline, it can be leaked in many ways, not necessarily over a cache covert channel (e.g., port contention [14] or execution unit timings [72]). It is beyond our scope to exhaustively evaluate all possible leaks to determine if a "confused" dereference can be exploited. Also, dereferences that appear unexploitable on our test setup may be exploitable with a different combination of kernel, compiler, and flags. Finally, today's unexploitable dereferences may become exploitable in the future, due to (1) discovery of new microarchitectural covert channels, (2) secrets reaching more instructions on future processors with deeper speculation windows, or (3) kernel code changes. Overall, the point is: *the architectural contract gets breached when the kernel dereferences a user-supplied address*. We thus focus on detecting these breaches.

Analysis We use the Radare2 framework [64] for static analysis and Triton [1] to perform taint tracking and symbolic execution. Conceptually, for each system call, we explore all possible execution paths that start at the system call's entry point and end with its return. We look for loads whose operand is unsafe (user-controlled) in one execution but safe (kernel-controlled) in another execution. We detect such loads by executing each path while performing taint analysis. We maintain a taint bit for each architectural register and each memory word. When analyzing a system call, we maintain two sets: U and K, initially empty. For each path, we initially taint the system call's arguments (which are user-controlled) and then symbolically execute the path while propagating taint as described below. When we encounter a load instruction, we place it into U if its operand is tainted, or into K otherwise. We flag every load $L \in U \cap K$. Listing 6 shows pseudo code of the algorithm.

Our analysis is designed for finding proofs of concept. As explained next, the analysis is not complete (may miss potential vulnerabilities), and it is not sound (may flag a load that is not a potential vulnerability). The results we report are after manually discarding such false positives.

Incomplete: We often cannot explore all possible execution flows of a system call, since their number is exponential in the number of basic blocks. We therefore skip some paths, which means we may miss potential vulnerabilities. We limit the number of paths analyzed in each system call in two ways.

First, we limit the number of explored paths but ensure that every basic block is covered. We start with the set *Paths* of the 1000 longest acyclic paths, to which we add the longest path $P \notin Paths$ that contains the basic block B, for each basic block B not already covered by some path in *Paths*. The motivation for adding these latter paths is to not ignore possible flows to *other* basic blocks; loads in a basic block covered by a single explored path cannot themselves be identified as vulnerable.

Second, when exploring paths, we do not descend into called functions (i.e., skip `call` instructions). Instead, we queue that called function and the taint state, and analyze them independently later. Overall, *our analysis not finding potential vulnerabilities does not imply that none exist.*

Unsound: The analysis is unsound because it abstracts memory and over-approximates taint. We model kernel and user memory as unbounded arrays, M^K and M^U, respectively. Let $T(x)$ denote the taint of x, where x is either a register or a memory location. Values in M^U are always tainted ($\forall a : T(M^U[a]) = 1$), whereas values in M^K are initially untainted, but may become tainted due to taint propagation. We execute a memory access with an untainted operand on M^K, and from M^U otherwise. A store $M^K[a] = R$ sets $T(M^K[a]) = T(R)$. A load $R = M^K[a]$ sets $T(R) = T(M^K[a])$, and similarly for loads from M^U. We assume that reads of uninitialized memory locations read 0. As a result, many objects in the analyzed execution appear to alias (i.e., occupy the same memory locations), which can lead to inaccurate taint

```
analyze_syscall(S):
  U = K = ∅
  G = control-flow graph of S
  for each acyclic path P ∈ G:
    analyze_path(P)

analyze_path(P):
  reset state, taint input argument registers
  for each instruction I in P:
    # propagate taint
    if I is a load/store: propagate taint from/to memory
    else: taint the output register of I iff
          one of its operands is tainted
    symbolically execute I
    if I is a load:
      add I to U or K, as appropriate
      flag I if I ∈ U ∩ K
```

Listing 6: Finding potential compiler-introduced speculative type confusion. (Conceptual algorithm, see text for optimizations.)

propagation. For instance, in the following code, a tainted value is stored into $M^K[0x10]$ and taints it, which causes the result of the subsequent load that reads $M^K[0x10]$ to be tainted. In the real execution, however, the store and load do not alias as they access different objects.

```
mov   %rax,0x10(%rbx)    # p->foo = x
mov   0x10(%rcx),%rdx    # v = q->bar
```

Due to its unsoundness, we manually verify every potential speculative type confusion flagged by the analysis. Because we limit the number of explored paths, the number of reports (and thus false positives) is not prohibitive to inspect.

5.3 Analysis results

We analyze Linux v5.4.11, compiled with GCC 5.8.2 and 9.3.0. We use the `allyes` kernel configuration, which enables every configuration option, except that we disable the kernel address sanitizer and stack leak prevention, as they instrument code and so increase the number of paths to explore. The case studies below are valid with these options enabled. We build the kernel with the `-Os` and `-O3` optimization levels. We analyze every system calls with arguments (393 in total). Table 4 summarizes the results.

Depending on the optimization level, GCC 9.3.0 introduces potential gadgets into 2–20 system calls, all of which stem from the same optimization (§ 5.3.1). GCC 5.8.2 introduces a "near miss" gadget into one system call (§ 5.3.2). This gadget is not exploitable in the kernel we analyze, but the pattern exhibited would be exploitable in other cases.

All the gadgets found are traditionally considered not exploitable, as the mispredicted branches depend on registers whose value is available (not cache misses), and so can resolve quickly. We show, however, that branches with available predicates *are* exploitable on certain x86 processors (§ 5.4).

compiler	flags	# vulnerable syscalls
GCC 9.3.0	-Os	20
GCC 9.3.0	-O3	2
GCC 5.8.2	-Os	0†
GCC 5.8.2	-O3	0

† One potential vulnerability exists, see § 5.3.2.

Table 4: Compiler-introduced speculative type confusion in Linux.

```
syscall(foo_t* uptr) {
  foo_t kfoo;
  // some code
  if (uptr)
    copy_from_user(&kfoo,
                   uptr,
                   ...);
  f(uptr ? &kfoo : NULL);
  // rest of code
}
```

(a) C pattern

```
# args: uptr in %rdi
  ...
  testq %rdi,%rdi
  je L # jump if %rdi==0
  # set copy_from_user args
  ...
  # %rbp contains addr of
  # stack buffer
  mov %rbp, %rdi
  call copy_from_user
L:callq f
```

(b) Emitted x86 code.

Listing 7: Reusing registers for a function call.

5.3.1 Supposedly NULL pointer dereference

The gadgets introduced by GCC 9.3.0 all stem from the same pattern, a simplified example of which appears in Listing 7. The system call receives an untrusted user pointer, `uptr`. If `uptr` is not NULL, it safely copies its contents into a local variable. Next, the system call invokes a helper `f`, which receives NULL if `uptr` was NULL, or a pointer to the kernel's local variable otherwise. The helper `f` (not shown) thus expects to receive a (possibly-NULL) kernel pointer, and therefore dereferences it (after checking it is non-NULL).

In the emitted code, the compiler introduces a speculative type confusion gadget by reusing `uptr`'s register to pass NULL to `f` when `uptr` is NULL. If the attacker invokes the system call with a non-NULL `uptr` and the NULL-checking branch mispredicts "taken" (i.e., `uptr`=NULL), then `f` gets called with the attacker-controlled value and dereferences it.

It is not clear that the gadget can be exploited, as both the mispredicted branch and the dereference depend on the same register. Why would the processor execute the dereference if it knows that the branch mispredicted? We discuss this in § 5.4.

5.3.2 Stack slot reuse

GCC 5.8.2 with the `-Os` (optimize for space) flag introduces an interesting gadget. The gadget instance we find is "almost" exploitable. We describe it not only to show how a small code change could render the gadget exploitable, but also to demonstrate how insidious a compiler-introduced gadget can be, and how difficult it is for programmers to reason about.

The gadget (Listing 8) is emitted into a function called by the `keyctl` system call. In this function, the compiler chooses to allocate space for the stack slot of a local variable (`dest_keyring`) by pushing the `rcx` register to the stack (a one-byte opcode) instead of subtracting from the stack pointer (a four-byte opcode). The `rcx` register holds a user-controlled value, one of the caller's (`keyctl`) arguments. The code then calls `get_instantiation_keyring()`, passing it the address of `dest_keyring`. If `get_instantiation_keyring()` returns successfully, the code calls `key_instantiate_and_link()`, which dereferences `dest_keyring`.

```
long keyctl_instantiate_key_common(key_serial_t id,
                                   struct iov_iter *from,
                                   key_serial_t ringid) {
 struct key *dest_keyring;
 // ... code ...
 ret = get_instantiation_keyring(ringid,rka,&dest_keyring);
 if (ret < 0)
  goto error2;
 ret = key_instantiate_and_link(rka->target_key, payload,
                                plen, dest_keyring,
                                instkey);
 // above call dereferences dest_keyring
}
```

(a) C code

```
 # %rcx is a live register from caller
 push %rcx
 # ... code ...
 lea    0x18(%r14),%rsi  # rka argument
 mov    %rsp,%rdx        # &dest_keyring argument
 mov    %r15d,%edi       # ringid argument
 callq  get_instantiation_keyring  # returns error
 test   %rax,%rax        # if (ret < 0)
 mov    %rax,%rbx
 js     error2           # mispredict no error
 ...
 mov    (%rsp),%rcx      # dest_keyring argument
 # dest_keyring could be old %rcx if not
 # overwritten in get_instantiation_keyring()
 callq  key_instantiate_and_link
```

(b) Emitted x86 code.

Listing 8: Attacker-controlled stack slot reuse in the `keyctl` system call flow.

In order to pass `dest_keyring` to `key_instantiate_and_link()`, the code reads its value from the stack. Consider what happens if the earlier `get_instantiation_keyring()` call encounters an error, and therefore leaves the stack slot with its original (user-controlled) value. In a correct execution, `key_instantiate_and_link()` never gets called, due to the error-checking flow. But if an attacker induces the error-checking branch to mispredict, `key_instantiate_and_link()` gets called with a user-controlled pointer argument to dereference. The only reason this instance is not exploitable is that `get_instantiation_keyring()` error flows overwrite `dest_keyring`. There is no security-related reason for this overwrite, since `dest_keyring` is a local variable that is never exposed to the user (i.e., there is no potential kernel information leak). Were this function to use a different coding discipline, the gadget would be potentially exploitable.

5.4 Potential exploitability of the gadgets

Exploiting the gadgets described in § 5.3 appears impossible. Typical Spectre gadgets exploit branches whose condition depends on values being fetched from memory, and so take a long time to resolve. In our case, the branch *still mispredicts*, since prediction is done at fetch time (§ 2.2). However, the branch condition is computable immediately when it enters the processor backend, as the values of the registers involved are known. One would expect the processor to immediately squash all instructions following the branch once it realizes that the branch is mispredicted, denying any subsequent leaking instructions (if they exist) a chance to execute.

The above thought process implicitly assumes that the processor squashes instructions in the shadow of a mispredicted branch when the branch is *executed*. But what if the squash happens only when the branch is *retired*? A branch's retirement can get delayed if it is preceded by a long latency instruction (e.g., a cache missing load), which would afford subsequent transient instructions a chance to execute.

We test the above hypothesis and find it to be false, but in the process, we discover that some x86 processors exhibit conceptually similar behavior due to other reasons.[7] Namely, we find that how x86 processors perform a branch misprediction squash depends in some complex way on *preceding branches* in the pipeline. Specifically, the squash performed by a mispredicting branch B_1 can get delayed if there is a preceding branch B_0 whose condition has not yet resolved.

Listing 9 shows the experiment. We test a gadget similar to the "supposedly NULL dereference" (§ 5.3.1) gadget. We train the victim so that both branches are taken (`*p==1`, `m!=bad_m`). We then invoke it so that both mispredict (`*p==0`, `m==bad_m`), with `p` being flushed from the cache, and test whether `m` is dereferenced and its value `s` is leaked. Table 5 shows the results: a leak can occur on both Intel and AMD processors, but its probability is minuscule on Intel processors. The small success probability and its dependence on the exact instructions in the gadget indicate that the leak occurs due some complex microarchitectural interaction.

Implications The fact that leaks can be realistically observed (for perspective: on AMD processors, our experiment observes $\approx 10\,\text{K}$ leaks per minute) means that compiler-introduced gadgets are a real risk. For any gadget instance, the kernel's flow may be such that there are slow-to-resolve branches preceding the gadget, and/or the attacker may be able to slow down resolution of preceding branches by evicting data from the cache.

[7]We did not test non-x86 processors.

```
int victim(int* p,
           T *m,
           T *bad_m,
           char *A) {
 if (*p == 1) {
  if (m != bad_m) {
   T s = *m;
   A[s*0x1000];
  }
  return 5;
 }
 return 0;
}
```

(a) C code

```
   # deref *p (cache miss)
   mov   (%rdi),%edi
   mov   $0x0,%eax
   cmp   $0x1,%edi   # *p==1 ?
   je    L2   # jmp if *p==1
L1:repz retq
L2:mov  $0x5,%eax
   cmp   %rdx,%rsi # m==bad_m ?
   je    L1   # jmp if m==bad_m
   movzbl (%rsi),%eax # s = *m
   shl   $0xc,%eax
   cltq
   add   %rax,%rcx
   movzbl (%rcx),%eax # leak s
   mov   $0x5,%eax
   jmp   L1
```

(b) Emitted x86 code (`T=char`).

Listing 9: Evaluating processor branch misprediction squashes.

processor	leak probability	
	T=char	**T=long**
AMD EPYC 7R32	$1/10^5$	$1/5000$
AMD Opteron 6376	$1/10^5$	$1/5000$
Intel Xeon Gold 6132 (Skylake)	$1/(5.09\times10^7)$	$1/(1.36\times10^6)$
Intel Xeon E5-4699 v4 (Broadwell)	$1/(3.64\times10^9)$	$1/(6.2\times10^9)$
Intel Xeon E7-4870 (Westmere)	$1/(1.67\times10^9)$	$1/(2.75\times10^7)$

Table 5: x86 branch squash behavior (in 30 B trials).

6 Speculative polymorphic type confusion

Linux defends from indirect branch target misspeculation attacks (Spectre-BTB) using *retpolines* [81]. A retpoline replaces an indirect branch with a thunk that jumps to the correct destination in a speculation-safe way, but incurs a significant slowdown [9]. Since the original Spectre-BTB attacks diverted execution to *arbitrary* locations, retpolines can appear as an overly aggressive mitigation, as they block all branch target speculation instead of restricting it to *legal* targets [9].

Accordingly, Linux is moving in the direction of replacing certain retpoline call sites with thunks of conditional *direct* calls to the call site's most likely targets, plus a retpoline fallback [24]. Listing 10 shows an example. JumpSwitches [9] take the idea further, and propose to dynamically promote indirect call sites into such thunks by learning probable targets and patching the call site online.

In this section, we detail how this approach can create speculative type confusion vulnerabilities (§ 6.1) and analyze the prevalence of such issues in Linux (§ 6.2)

```
# %rax = branch target
  cmp $0xXXXXXXXX, %rax  # target1?
  jz $0xXXXXXXXX
  cmp $0xYYYYYYYY, %rax  # target2?
  jz $0xYYYYYYYY
  ...
  jmp ${fallback} # jmp to retpoline thunk
```

Listing 10: Conditional direct branches instead of indirect branch.

6.1 Virtual method speculative type confusion

It has been observed (in passing) that misprediction of an indirect call's target can lead to speculative type confusion in object-oriented polymorphic code [18]. The problem occurs when a branch's valid but wrong target, f, is speculatively invoked instead of the correct target, g. The reason that both f and g are valid targets is that both implement some common interface. Each function, however, expects some or all of its arguments to be a different *subtype* of the types defined by the interface. As a result, when the misprediction causes f's code to run with g's arguments, f might *derive* a variable v of type T_f from one of g's arguments, which is really of type T_g.

Prior work [32, 56] describes how, if T_g is smaller than T_f, f might now perform an out-of-bounds access. We observe, however, that the problem is more general. Even if both types are of the same size, f might still dereference a field in T_f which now actually contains some user-controlled content from a field in T_g, and subsequently inadvertently leak the read value. Moreover, the problem is transitive: f might dereference a field that is a pointer in both T_f and T_g, but points to an object of a different type in each, and so on.

In the following sections, we analyze the prevalence of potential polymorphism-related speculative type confusion is Linux. Our analysis is forward looking: we explore *all* indirect call sites, not only the ones that have already been converted to conditional call-based thunks. Such a broad analysis helps answering questions such as: How likely is it that manually converting a call site (the current Linux approach) will create a vulnerability? What are the security implications of adopting an "all-in" scheme likes JumpSwitches?

6.2 Linux analysis

Linux makes heavy use of polymorphism and data inheritance (subtype derivation) in implementing internal kernel interfaces. (Linux code implements inheritance manually, due to being in C, as explained below.) We perform a source-level vulnerability analysis by extending the smatch static analysis tool [20]. As in § 5, we do not claim that if our analysis finds no vulnerabilities then none exist.

At a high-level, the analysis consists of four steps, detailed below. (Listing 11 shows pseudo code.)

① **Find legal targets:** For each structure type and each function pointer field in that type, we build a set of legal targets that this field might point to.

```
# targets: a mapping from function pointer
# fields in types to their valid targets
# derivedObjs: a mapping from function
# arguments to possible private structs they
# derive
 # ① find call site target
 for every assignment $x.a = g$ where $g$ is a function
     and $x$ is of type $T$
  targets[$T,a$].add($g$)
 # ② find derived_objects
 for every $g$ in targets, scan control-flow graph of $g$:
  if $i$-th arg of $g$ is used to derive struct of type $T$:
   derivedObjs[$g,i$] = $T$
 # ③ find all overlaps
 overlaps = set()
 for every $T,a \mapsto \{g_1,\ldots,g_m\}$ in targets:
  for every pair $(g_i,g_j)$:
   for every $g_i,a \mapsto D_i$ in derivedObjs:
    for every field $f_i$ of $D_i$ that is user-controllable:
     $D_j$ = derivedObjs[$g_j,a$]
     let $f_j$ be the overlapping field in $D_j$
     overlaps.add(($g_i,D_i,f_i,g_j,D_j,f_j$))
 # ④ find potentially exploitable overlaps
 for each $(g_i,D_i,f_i,g_j,D_j,f_j)$ in overlaps:
  scan control-flow graph of $g_j$
  if $D_j.f_j$ is dereferenced:
   let v be the data read from $D_j.f_j$
   if v is used to index an array or v is dereferenced:
     flag $(g_i,D_i,f_i,g_j,D_j,f_j)$
```

Listing 11: Finding potential speculative polymorph type confusion.

② **Identify subtype derivations:** For each function g that is a legal target of some call site, we attempt to identify the arguments used to derive g-specific (subtype) objects. Since Linux implements data inheritance manually, we scan for the relevant patterns (illustrated in Listing 12): (1) a "private data" field in the parent structure points to the derived object (Listing 12a); (2) the derived object is the first field in the parent, and obtained by casting (Listing 12b); and (3) the derived object is some field in the parent, extracted using the container_of macro (Listing 12c).

③ **Find overlapping fields:** This is a key step. For every pair of functions that are legal targets of some call site, we search for *overlapping* fields among the objects derived from the same function argument. Two fields overlap if (1) their (start,end) offset range in the respective object types intersect, (2) one field is user-controllable, and (3) the other field, referred to as the *target*, is not user-controllable. We rely on smatch to identify which fields are user-controllable, which is done based on Linux's __user annotation [80] and heuristics for tracking untrusted data, such as network packets. An overlap where the target field is a kernel pointer can potentially lead to an attacker-controlled dereference.

④ **Search for vulnerabilities:** This steps takes a pair of functions g_i, g_j identified as using derived objects with overlapping fields, and tries to find if the overlaps are exploitable. We run a control- and data-flow analysis on g_j, the function

```
struct Common {
 void* private;
};
struct Derived {...};

void foo(Common* c) {
 Derived* d = c->private;
```

(a) Private field.

```
struct Common {...}
struct Derived {
 struct Common common_data;
 ...
}
void foo(Common* c) {
 Derived* d = (Derived*) c;
```

(b) Casting.

```
struct Derived {
 ...
 struct Common common;
 ...
}
void foo(Common* c) {
 Derived* d = container_of(c, Derived*, common);
```

(c) Contained structs.

Listing 12: Linux data inheritance patterns.

using the object with the target field, and check if that field is dereferenced. This process finds thousands of potential attacker-controlled dereferences. To make manual verification of the results tractable, we try to detect if the value read by the dereference gets leaked via a cache covert channel. We consider two types of leaks: if some array index depends on the value, and a "double dereference" pattern, in which the value is a pointer that is itself dereferenced. The latter pattern can be used to leak the L1-indexing bits of the value.

6.3 Analysis results

We analyze Linux v5.0-rc8 (default configuration) and v5.4.11 (`allyes` configuration). Table 6 summarizes the results. While we find thousands of potential attacker-controlled dereferences, most are double dereferences, which we do not consider further. Manual inspection of the array indexing cases reveals that they are *latent* vulnerabilities, which are not (or likely not) exploitable, but could become exploitable by accident:

- Most cases let the attacker control <64 bits of the target pointer, with which it cannot represent a kernel pointer (e.g., attacker controls a 64-bit field in its structure, but it only overlaps the target field over one byte). A change in structure layout or field size could make these cases exploitable.
- In other cases, the attacker does not have full control over its field (e.g., it is a flag set by userspace that can only take on a limited set of values). § 6.4 shows an example of such a case. A change in the semantics of the field could render these cases exploitable.
- Some cases are false positives due to imprecision of the analysis (e.g., a value read and used as an array index is masked, losing most of the information).

	5.0-rc8 (def.)	**5.4.11 (allyes)**
# flagged	2706	8814
double deref	2578	8512
array indexing	128	302
array indexing exploitable?		
no: <64 bit overlap	108	261
no: limited control	11	13
no: other	4	17
no(?): speculation window	5	11

Table 6: Potential speculative polymorph data confusion in Linux.

- Finally, we discard some cases because there is a function call between the dereference and the array access, so we assume that the processor's speculation window in insufficient for the value to reach the array access.

Note that we may be over-conservative in rejecting cases. The reason that we do not invest in exploring each case in detail is because we are looking at an analysis of all indirect calls, most of which are currently protected with retpolines. The takeaway here is that were a conditional branch-based mitigation used instead of retpolines, the kernel's security would be on shaky ground.

6.4 Case study of potential vulnerability

To get a taste for the difficulty of reasoning about this type of speculative type confusion, consider the example in Listing 13. The functions in question belong the USB type C driver and to the `devfreq` driver. They implement the `show` method of the driver's attributes, which is used to display the attributes in a human-readable way. Both functions extract a derived object from the first argument using `container_of`. The attacker trains the call site to invoke the USB driver's method (Listing 13a) by repeatedly displaying the attributes of that device. Next, the attacker attempts to display the `devfreq` driver's attributes. Due to the prior training, instead of the `devfreq` method (Listing 13b) being executed, the USB's method is initially speculatively executed. Consequently, the USB method's derived object actually points to `devfreq`'s object, so when the USB method dereferences its `cap` field it is actually dereferencing the value stored in the `devfreq`'s structure `max_freq` field. However, as shown in Listing 13c, the attacker can only get `max_freq` to contain one of a fixed set of values. A similar scenario, in which `max_freq` would be some 64-bit value written by the user, would be exploitable.

7 Discussion & mitigations

Here, we discuss possible mitigations against speculative type confusion attacks. We distinguish mitigations for the general problem (§ 7.1) from the specific case of eBPF (§ 7.2). We focus on immediately deployable mitigations, i.e., mainly software mitigations. Long term defenses are discussed in § 8.

```
ssize_t port_type_show(struct device *dev,
                       struct device_attribute *attr,
                       char *buf) {
 // container_of use
 struct typec_port *port = to_typec_port(dev);
 if (port->cap->type == TYPEC_PORT_DRP)
  return ...;
 return sprintf(buf, "[%s]\n",
         typec_port_power_roles[port->cap->type]);
}
```

(a) Mispredicted target.

```
ssize_t max_freq_show(struct device *dev,
                      struct device_attribute *attr,
                      char *buf) {
 // container_of use
 struct devfreq *df = to_devfreq(dev);
 return sprintf(buf, "%lu\n", min(df->scaling_max_freq,
                                  df->max_freq));
}
```

(b) Actual target.

```
ssize_t max_freq_store(struct device *dev,
                       struct device_attribute *attr,
                       const char *buf, size_t count) {
 ...
 if (freq_table[0] < freq_table[df->profile->max_state - 1])
  value = freq_table[df->profile->max_state - 1];
 else
  value = freq_table[0];
  ... value is stored into max_freq ...
}
```

(c) Value of `max_freq` is constrained.

Listing 13: Speculative polymorphic type confusion case study.

7.1 General mitigations

Unlike bounds check bypass gadgets, speculative type confusion gadgets do not have a well-understood, easy to spot structure, and are difficult if not impossible for programmers to reason about. Mitigating them thus requires either complete Spectre protection or statically identifying every gadget and manually protecting it.

Complete mitigations Every Spectre attack, including speculative type confusion, can be fully mitigated by placing speculation barriers or serializing instructions after every branch. This mitigation essentially disables speculative execution, leading to huge performance loss [18]. Speculative load hardening (SLH) [21] (implemented in Clang/LLVM [48] and ICC [38]) is a more efficient complete mitigation. SLH does not disable speculative execution, but only blocks results of speculative loads from being forwarded down the pipeline until the speculative execution proves to be correct. To this end, SLH masks the output of every load with a mask that has a data dependency on the outcome of *all* prior branches in the program, which is obtained by emitting conditional move instructions that maintain the mask after every branch.

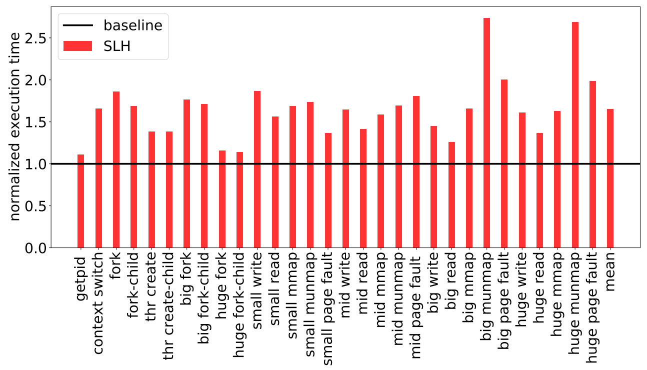


Figure 3: Slowdown of Linux 5.4.119 kernel operations due to SLH.

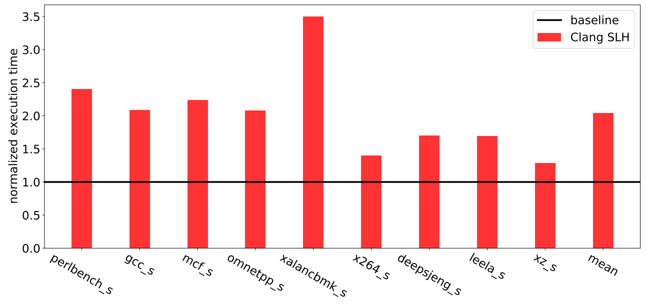


Figure 4: Slowdown of SPEC CPU 2017 applications due to SLH.

Unfortunately, we find that SLH imposes significant overhead on both kernel operations and computational workloads, far worse than previously reported results on a Google microbenchmark suite [21]. To evaluate SLH's overhead on the Linux kernel, we use LEBench [65], a microbenchmark suite that measures the performance of the most important system calls for a range of application workloads.[8] To evaluate SLH's overhead on computational userspace workloads, we use the SPEC CPU2017 benchmark suite [17].

We evaluate the above benchmarks on a system with a 2.6 GHz Intel Xeon Gold 6132 (Skylake) CPU running Linux 5.4.119. Figure 3 shows the relative slowdown of system call execution time with an SLH-enabled kernel, compared to a vanilla kernel (which includes Linux's standard Spectre mitigations but is compiled without SLH). Figure 4 shows the relative execution time slowdown of a subset of the CPU2017 C/C++ benchmarks when compiled with SLH enabled, com-

[8]We modify Clang/LLVM's SLH implementation to support kernel-mode execution, which is not supported out of the box. SLH assumes that the high bits of addresses are zeros, and relies on this property to encode information in the stack pointer on function calls/returns [21]. This technique breaks kernel execution, because the high bits of kernel virtual addresses are ones. Our modification simply flips the bit values that SLH encodes in the stack pointer, so that non-transient kernel executions maintain a valid stack pointer.

pared to with standard compilation. In both settings, SLH imposes significant slowdowns. SLH causes an average system call slowdown of 1.65× (up to 2.7×) and an average CPU2017 program slowdown of 2× (up to almost 3.5×).

Other proposed software mitigations [62, 87] use similar principles to SLH, but were evaluated on protecting array bounds checking. It is not clear what their overhead would be if used for complete mitigation.

Spot mitigations We contend that manual Spectre mitigation, as advocated in Linux and GCC, is not practical against speculative type confusion. Similarly to how transient execution attacks break the security contract between hardware and software, speculative type confusion breaks the contract between the compiler and programs, with correct programs possibly being compiled into vulnerable native code. Worse, any conclusion reached about security of code can be invalidated by an unrelated code change somewhere in the program or an update of the compiler. Overall, human-only manual mitigation seems difficult if not infeasible.

As a result, a manual mitigation approach must be guided by a complete static analysis, which would detect every speculative type confusion gadget in the kernel. It is notoriously difficult, however, to *prove* safety of C/C++ code, e.g., due to pointer aliasing and arithmetic [12]. Here, the problem is compounded by the need to analyze all possible paths, which invalidates many static analysis optimizations. Indeed, current analyses that reason about speculative execution vulnerabilities have limited scalability [31], restrict themselves to constant-time code [22], or search for specific syntactic code patterns [88]. Scaling an analysis to *verify* that every pointer dereference in Linux is safe from speculative type confusion is a major research challenge.

Hardware workarounds Using different BPUs for user and kernel context may be a non-intrusive hardware change that vendors can quickly roll out. However, this mitigation would still allow attackers to perform mistraining by invoking in-kernel shadow branches (e.g., in eBPF programs) whose PHT entries collide with the victim's.

7.2 Securing eBPF

In addition to the generic mitigations, eBPF can defend from speculative type confusion in eBPF-specific ways. The verifier can reason about all execution flows, not just semantically correct ones. However, this approach would increase verification time and render some programs infeasible to verify. An alternative approach is for the verifier to inject masking instructions to ensure that the operand of *every* load instruction is constrained to the object it is supposed to be accessing, generalizing the sandboxing approaches of Native Client x86-64 [73] and Swivel-SFI [61].

8 Related work

Attacks Blindside [30] and SpecROP [13] employ Spectre-PHT attacks that do not involve a bounds check bypass. Both attacks also involve indirect branching to an illegal target, whereas our exploitation of indirect branches does not. Blindside leverages a non-speculative pointer corruption (e.g., via a buffer overflow) to speculatively hijack control flow in the shadow of a mispredicted branch. SpecROP poisons the BTB to chain multiple Spectre gadgets with indirect calls. With recent mitigations, SpecROP is therefore limited to intra-address space attacks and cannot target the kernel.

Defenses Non-speculative type confusion [33, 40, 49, 60] and control-flow integrity (CFI) [6,25,28,78,96] have received significant attention. These works generally consider non-speculative memory corruption and control-flow hijacking, not memory disclosure over covert channels. The defenses proposed are based on the architectural semantics, and so do not straightforwardly apply to speculative execution attacks.

There are many proposals for hardware defenses against transient execution attacks. Some designs require programmer or software support [27, 42, 69, 77, 93] but many are software-transparent. Transparent designs differ in the protection approach. Some block only cache-based attacks [7, 41, 50, 66, 67, 91], whereas others comprehensively block data from reaching transient covert channels [10, 89, 94, 95]. These works all report drastically lower overhead than what we observe for SLH, but their results are based on simulations.

Combining the above two lines of work, SpecCFI [47] is a hardware mitigation for Spectre-BTB (v2) attacks that restricts branch target speculation to legal targets, obtained by CFI analysis. SpecCFI also assumes hardware Spectre-PHT (v1) mitigations, and thus should not be vulnerable to speculative type confusion.

In principle, speculative type confusion can be detected by static [22, 31, 88] or dynamic [63] analysis that reasons about speculative execution. To our knowledge, only SPECTECTOR [31] performs a sound analysis targeting general-purpose code, but it has challenges scaling to large code bases, such as Linux. Other static analyses target only constant-time code [22] or search for specific code patterns [88]. SpecFuzz [63] dynamically executes misspeculated flows, making them observable to conventional memory safety checkers, such as AddressSanitizer [74]. Thus, SpecFuzz is not guaranteed to find all vulnerabilities.

eBPF Gershuni et al. [29] leverage abstract interpretation to design an eBPF verifier with improved precision (fewer incorrectly rejected programs) and scope (verifying eBPF programs with loops). Their analysis still is based on architectural semantics, and thus does not block our described speculative type confusion attack.

9 Conclusion

We have shown that speculative type confusion vulnerabilities exist in the wild. Speculative type confusion puts into question "spot" Spectre mitigations. The relevant gadgets do not have a specific structure and can insidiously materialize as a result of benign compiler optimizations and code changes, making it hard if not impossible for programmers to reason about code and manually apply Spectre mitigations.

Speculative type confusion vulnerabilities also slip through the cracks of non-comprehensive Spectre mitigations such as prevention of bounds check bypasses and restriction of indirect branch targets to legal (but possibly wrong) targets. Consequently, the Spectre mitigation approach in the Linux kernel—and possibly other systems—requires rethinking and further research.

Disclosure

We disclosed our findings to the Linux kernel security team, the eBPF maintainers, as well as Google's Android and Chromium teams in June 2020. Following our report, Google awarded us a Vulnerability Reward. The eBPF vulnerability (CVE-2021-33624) was fixed in the mainline Linux development tree in June 2021, by extending the eBPF verifier to explore speculative paths [16]. Subsequently, we issued an advisory [44] to alert the various Linux distributions to the vulnerability and its mitigation.

Acknowledgements

We thank Alla Lenchner for extending LLVM's SLH to support kernel-mode mitigation. We thank the reviewers and our shepherd, Deian Stefan, for their insightful feedback. This work was funded in part by an Intel Strategic Research Alliance (ISRA) grant and by the Blavatnik ICRC at TAU.

References

[1] *Triton: A Dynamic Symbolic Execution Framework*, 2015.

[2] eXpress Data Path. https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/index.html, 2018.

[3] IO Visor Project. https://www.iovisor.org/technology/bcc, 2018.

[4] A seccomp overview. https://lwn.net/Articles/656307/, 2018.

[5] Linux kernel virtual memory map. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, 2020.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *CCS*, 2005.

[7] Sam Ainsworth and Timothy M. Jones. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In *ISCA*, 2019.

[8] AMD. An Update on AMD Processor Security. https://www.amd.com/en/corporate/speculative-execution-previous-updates#paragraph-337801, 2018.

[9] Nadav Amit, Fred Jacobs, and Michael Wei. JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre. In *USENIX ATC*, 2019.

[10] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels. In *PACT*, 2019.

[11] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture. In *SIGCOMM*, 1999.

[12] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *CACM*, 53(2), 2010.

[13] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M. Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. SpecROP: Speculative Exploitation of ROP Chains. In *RAID*, 2020.

[14] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS*, 2019.

[15] Daniel Borkmann. bpf: prevent out of bounds speculation on pointer arithmetic. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=979d63d50c0c0f7bc537bf821e056cc9fe5abd38, 2019.

[16] Daniel Borkmann. bpf: Fix leakage under speculation on mispredicted branches. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9183671af6dbf60a1219371d4ed73e23f43b49db, 2021.

[17] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *ICPE*, 2018.

[18] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*, 2019.

[19] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *CCS*, 2020.

[20] Dan Carpenter. Smatch!!! http://smatch.sourceforge.net, 2003.

[21] Chandler Carruth. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[22] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-Time Foundations for the New Spectre Era. In *PLDI*, 2020.

[23] CodeMachine. Windows Kernel Virtual Address Layout. https://www.codemachine.com/article_x64kvas.html, 2020.

[24] Jonathan Corbet. Relief for retpoline pain. LWN (https://lwn.net/Articles/774743/), 2018.

[25] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *IEEE S&P*, 2014.

[26] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[27] Jacob Fustos, Farzad Farshchi, and Heechul Yun. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. *DAC*, 2019.

[28] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE Euro S&P*, 2016.

[29] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *PLDI*, 2019.

[30] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, 2020.

[31] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *IEEE S&P*, 2020.

[32] Noam Hadad and Jonathan Afek. Overcoming (some) Spectre browser mitigations. https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/, 2018.

[33] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical Type Confusion Detection. In *CCS*, 2016.

[34] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 6th edition, 2017.

[35] Jann Horn. Issue 1711: Linux: eBPF Spectre v1 mitigation is insufficient. https://bugs.chromium.org/p/project-zero/issues/detail?id=1711, 2018.

[36] Jann Horn. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[37] Intel. Bounds Check Bypass / CVE-2017-5753 / INTEL-SA-00088. https://software.intel.com/security-software-guidance/software-guidance/bounds-check-bypass, 2018.

[38] Intel. Code Generation Options: mconditional-branch, Qconditional-branch. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/compiler-option-details/code-generation-options/mconditional-branch-qconditional-branch.html, 2020.

[39] Brian Krzanich (Intel). Advancing Security at the Silicon Level. https://newsroom.intel.com/editorials/advancing-security-silicon-level/, 2018.

[40] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient Detection of Type Confusion Errors for C++. In *CCS*, 2017.

[41] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *DAC*, 2019.

[42] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO*, 2018.

[43] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv e-prints*, 1807.03757, 2018.

[44] Ofek Kirzner and Adam Morrison. CVE-2021-33624: Linux kernel BPF protection against speculative execution attacks can be bypassed to read arbitrary kernel memory. https://www.openwall.com/lists/oss-security/2021/06/21/1, 2021.

[45] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*, 2019.

[46] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[47] Esmaeil Mohammadian Koruyeh, Shirin Hajl Amin Shirazi, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. SPECCFI: Mitigating Spectre Attacks Using CFI Informed Speculation. In *IEEE S&P*, 2020.

[48] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[49] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type Casting Verification: Stopping an Emerging Attack Vector. In *USENIX Security*, 2015.

[50] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *HPCA*, 2019.

[51] Linux. Mitigating speculation side-channels. https://www.kernel.org/doc/Documentation/speculation.txt, 2018.

[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[53] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*, 2015.

[54] Giorgi Maisuradze and Christian Rossow. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[55] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv e-prints*, 1902.05178, 2019.

[56] Microsoft. C++ Developer Guidance for Speculative Execution Side Channels. https://docs.microsoft.com/en-us/cpp/security/developer-guidance-speculative-execution?view=vs-2019, 2018.

[57] Microsoft. Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, 2018.

[58] Microsoft. /Qspectre. https://docs.microsoft.com/en-us/cpp/build/reference/qspectre-load?view=vs-2019, 2019.

[59] Microsoft. /Qspectre-load. https://docs.microsoft.com/en-us/cpp/build/reference/qspectre-load?view=vs-2019, 2020.

[60] Paul Muntean, Sebastian Wuerl, Jens Grossklags, and Claudia Eckert. CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM. In *ESORICS*, 2018.

[61] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security Symposium*, 2021.

[62] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *arXiv e-prints*, arXiv:2005.00294, 2018.

[63] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *USENIX Security*, 2020.

[64] Pancake. Radare2. https://rada.re/n/.

[65] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An Analysis of Performance Evolution of Linux's Core Operations. In *SOSP*, 2019.

[66] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An "Undo" Approach to Safe Speculation. In *MICRO*, 2019.

[67] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *ISCA*, 2019.

[68] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt, 2018.

[69] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: Leakage-Free Transient Execution. In *NDSS*, 2020.

[70] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[71] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*, 2017.

[72] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS*, 2019.

[73] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security*, 2010.

[74] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*, 2012.

[75] Alexei Starovoitov. bpf: enable non-root eBPF programs (Linux 4.4 commit), 2015.

[76] Alexei Starovoitov. bpf: prevent out-of-bounds speculation. https://lwn.net/Articles/743288/, 2018.

[77] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *ASPLOS*, 2019.

[78] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security*, 2014.

[79] Robert M Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, (1), 1967.

[80] Linus Torvalds. Add __user__kernel address space modifiers. https://lwn.net/Articles/28348/, 2003.

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

[82] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[83] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE S&P'20*, 2020.

[84] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, , and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security*, 2018.

[85] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE S&P*, 2019.

[86] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *IEEE S&P*, 2021.

[87] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. In *POPL*, 2021.

[88] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *IEEE Transactions on Software Engineering*, 2019.

[89] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas Wenisch, and Baris Kasikci. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO*, 2019.

[90] Wikipedia. Usage share of operating systems. https://en.wikipedia.org/wiki/Usage_share_of_operating_systems#Public_servers_on_the_Internet, 2020.

[91] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO*, 2018.

[92] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[93] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS*, 2019.

[94] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. Speculative Data-Oblivious Execution (SDO): Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *ISCA*, 2020.

[95] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO*, 2019.

[96] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security*, 2013.

[97] To Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *ASPLOS*, 2020.