VeriFence: Lightweight and Precise Spectre Defenses for Untrusted Linux Kernel Extensions

Luis Gerhorst
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
Germany
gerhorst@cs.fau.de

Henriette Herzog
Ruhr-Universität Bochum (RUB)
Germany
henriette.herzog@rub.de

Peter Wägemann
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
Germany
waegemann@cs.fau.de

Maximilian Ott
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
Germany
ott@cs.fau.de

Rüdiger Kapitza
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)
Germany
ruediger.kapitza@fau.de

Timo Hönig
Ruhr-Universität Bochum (RUB)
Germany
timo.hoenig@rub.de

ABSTRACT

High-performance IO demands low-overhead communication between user- and kernel space. This demand can no longer be fulfilled by traditional system calls. Linux's extended Berkeley Packet Filter (BPF) avoids user-/kernel transitions by just-in-time compiling user-provided bytecode and executing it in kernel mode with near-native speed. To still isolate BPF programs from the kernel, they are statically analyzed for memory- and type-safety, which imposes some restrictions but allows for good expressiveness and high performance. However, to mitigate the Spectre vulnerabilities disclosed in 2018, defenses which reject potentially-dangerous programs had to be deployed. We find that this affects 31 % to 54 % of programs in a dataset with 844 real-world BPF programs from popular open-source projects. To solve this, users are forced to disable the defenses to continue using the programs, which puts the entire system at risk.

To enable *secure and expressive* untrusted Linux kernel extensions, we propose VeriFence, an enhancement to the kernel's Spectre defenses that reduces the number of BPF application programs rejected from 54 % to zero. We measure VeriFence's overhead for all mainstream performance-sensitive applications of BPF (i.e., event tracing, profiling, and packet processing) and find that it improves significantly upon the status-quo where affected BPF programs are either unusable or enable transient execution attacks on the kernel.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Side-channel analysis and countermeasures**; • **Software and its engineering** → *Automated static analysis*; Software safety; **Just-in-time compilers**.

KEYWORDS

eBPF, Transient Execution Attacks, High-Performance Networking

ACM Reference Format:
Luis Gerhorst, Henriette Herzog, Peter Wägemann, Maximilian Ott, Rüdiger Kapitza, and Timo Hönig. 2024. VeriFence: Lightweight and Precise Spectre Defenses for Untrusted Linux Kernel Extensions. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30–October 02, 2024, Padua, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3678890.3678907

1 INTRODUCTION

Safe kernel extensions as implemented by Linux extended Berkeley Packet Filter (BPF) offer very low-overhead interaction between user and kernel space. Their applications include network input/output (IO) [62, 98, 120], memory optimization [80], threat detection [72], isolation [70], tracing [128], scheduling [58], and storage [133]. While most applications today require root privileges (*privileged BPF*), BPF also allows unprivileged user processes to load safety-checked bytecode into the kernel (*unprivileged BPF*), which executes at near-native speed. Users typically develop BPF programs in a high-level programming language (e.g. C or Rust), which is compiled into BPF bytecode. This bytecode is verified in regard to its safety before being just-in-time compiled. Invoking BPF programs in the kernel and calling kernel functions from within BPF is much faster than the respective switch to/from user context [49]. However, this performance advantage comes at a cost: Executing untrusted code in the kernel address space requires precise static analysis of the untrusted program to maintain isolation. While some advanced safety-checks designed to prevent transient execution attacks can be omitted for privileged BPF, this is not possible for unprivileged BPF. Ensuring that these advanced defenses are functional and low-overhead is therefore mandatory in order to enable the various potential use cases for unprivileged BPF (e.g., network traffic filters [7], io_uring [16], Seccomp [70], and others [6, 43]).

While Meltdown and similar domain-bypass transient execution attacks (e.g., where the hardware crosses address-space boundaries [2]) can be efficiently prevented in hardware, the cross/in-domain transient execution attacks discovered in 2018 [76] still

---

RAID 2024, September 30–October 02, 2024, Padua, Italy

ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678907

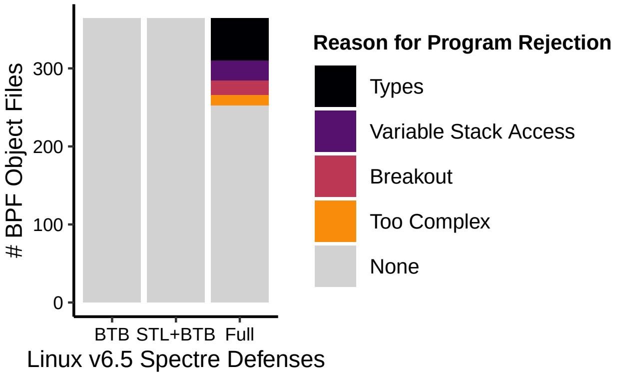

Figure 1: Reasons for program rejection in Linux when enabling defenses against transient execution attacks for 364 real-world BPF test, example, and application object files. While the Spectre-*BTB* and -*STL* defenses do not trigger program rejections, enabling Spectre-PHT defenses (*Full*, i.e., PHT+STL+BTB) prevents 31 % **of all objects (and even** 54 % **of application objects) from being used.**

require software defenses on all high-performance central processing units (CPUs) [63, 87] to date. Two cross/in-domain transient execution attacks are a particular challenge for BPF: Spectre-PHT exploits the Pattern History Table's potential for branch misprediction, and Spectre-STL exploits speculative store bypass (via the CPU's Store To Load buffer). Branches and stores are common in BPF bytecode, but malicious programs can use these very instructions to break out of their sandbox using transient execution. For example, following a mispredicted branch, the program can perform an unchecked out-of-bounds (OOB) access and leak the resulting kernel secret using a covert channel. Preventing these attacks entirely in hardware requires far-reaching changes to processor designs with significant performance impacts [126] (e.g., 9 % to 15 % for [131]), that high-performance CPU vendors to date have not implemented [8–10].

However, developing practical Spectre-resilient sandboxes is an open research problem. To the best of our knowledge, BPF is the only widely deployed sandbox that attempts to implement full software-defenses against Spectre-PHT, Spectre-BTB, and Spectre-STL. All other widely deployed sandboxes such as Java [91], WebAssembly (Wasm) [41, 122], and JavaScript [87, 102] instead rely on process isolation. This is the approach recommended by Intel [63] and V8 developers [31], but it is not applicable to BPF.

To implement the sandbox, the kernel statically analyzes the control and data flow of the BPF programs before allowing them to execute in its address space. Traditionally, the kernel only considered BPF program paths that could actually execute architecturally during this static analysis. In response to the Spectre-PHT and Spectre-STL vulnerabilities disclosed in 2018, kernel developers have extended the BPF verifier [5] to prevent leaks from transiently-executed BPF program paths. This includes (a) inserting instructions to make speculation safe (e.g., index masking) [25, 86, 114], (b) inserting instructions to prevent speculation (e.g., x86-64 `lfence`) [27], or (c) statically analyzing the behavior on speculative code paths to ensure they are safe, in turn rejecting the program entirely if it exhibits any unsafe behavior [28]. The latter is required to prevent Spectre-PHT, but it severely limits BPF because it prevents the user from loading their extension into the kernel altogether.

To analyze the Spectre-PHT defenses' impact on BPF, we collect 364 BPF object files (each containing one or multiple BPF programs) from open-source projects and enable defenses. The results, shown in Figure 1, confirm that the number of rejections is significant and are further discussed in Section 5.3. Restructuring the program to not exhibit unsafe transient behavior is tedious, as users are usually not familiar with transient execution vulnerabilities and only indirectly control the compiler-generated bytecode in their high-level sources.

Solving this problem of pessimistically rejected unprivileged programs would make numerous new applications of BPF practical. This is because everything-but-trivial programs (approximately 10s of lines of C code) are currently easily rejected, making development extremely tedious. It is our main motivation to allow for more powerful and easier-to-develop unprivileged programs and thereby enable the numerous future use-cases for unprivileged BPF that have been proposed by upstream kernel developers and academia:

- **Network Traffic Filters:** In upstream Linux, unprivileged BPF is already allowed to filter network traffic [7]. Resolving the rejection issue will allow for more precise and easier-to-develop filters. These are extremely useful as demonstrated by the high number of projects that already utilize such filters [62, 98, 120] but currently require root privleges.
- **io_uring:** This will enable processes to utilize even better-performing asynchronous I/O due to a reduced number of switches to userspace [16].
- **Seccomp:** This will allow unprivileged users to start processes that are restricted in regard to system calls and system-call parameters. In comparison to the existing solutions, it will allow for more precise filtering and thereby reduce the system's attack surface [70].

Aside from these three applications, there are various others [6, 43] as unprivileged, safe kernel extensions are an extremely generic tool. All these applications of unprivileged BPF can only become practical if the issues (i.e., program rejections) induced by the Spectre defenses are resolved.

To solve this problem of pessimistically rejected unprivileged BPF programs, our work extends the upstream BPF verifier [5] by implementing VeriFence, an improved defense approach for Spectre. The core benefit of VeriFence is that it reduces the number of programs that cannot be automatically mitigated and are, thus, rejected by the state-of-the-art BPF verifier.

Our main contributions are fivefold:

- **Domain Analysis**: To inform our design decisions, we statically analyze 844 real-world BPF programs from six popular software projects regarding their code size and complexity. This domain analysis gives us a detailed picture of the BPF program landscape.
- **Security Notions:** Real-world Spectre defenses lack behind the theoretical foundations [73]. We discuss BPF's security

in the light of transient execution attacks using established speculative security properties from the literature.
- **Design:** We design the VeriFence defense which optimistically attempts to verify all speculative execution paths and only falls back to speculation barriers when unsafe behavior is detected.
- **Evaluation**: We evaluate the performance impact of VeriFence on the three most common use cases for BPF: event tracing, continuous profiling, and network load balancing. We find that VeriFence is lightweight, as it does not increase BPF's invocation latency.
- **Implementation**: We publish VeriFence for the v6.5 Linux kernel. Our patches soundly combine speculation barriers and static analysis to not increase the kernel's attack surface.

Besides our five main contributions, we discover multiple Linux kernel bugs during our analyses. Based on our findings, we contribute proof-of-concept exploits [47] and fixes [48, 50] that have been accepted into upstream Linux.

2 BACKGROUND

This section discusses the transient execution vulnerabilities relevant to BPF and presents the respective defenses in Linux v6.5. We focus on cross/in-domain transient execution attacks [2] but not on domain-bypass transient execution attacks (including Meltdown / Spectre v3 [81], MDS [106], L1TF [30]) because they can be efficiently addressed in hardware and are not specific to BPF. We focus on the **Spectre-PHT (v1), -STL (v4), and -BTB (v2)** attacks because these are the longest-standing vulnerabilities to which new processors are still vulnerable [8–10].

2.1 Transient Execution Vulnerabilities

While regular (architectural) timing side-channel attacks can happen only when the program computes explicitly on sensitive data (e.g., cryptographic keys, or also kernel pointers with kernel address space layout randomization, KASLR), timing side-channel attacks based on transient execution can also target victim code that does not explicitly work with sensitive data. They become possible whenever a victim program encodes secrets into side channels (e.g., the cache) during transient execution.

2.1.1 Speculation Triggers. Transient execution CPU vulnerabilities are commonly grouped by the microarchitectural component that speculates. For BPF, Spectre-PHT, -STL, and -BTB are of particular relevance [32, 36]. Spectre-PHT [76] includes all transient execution vulnerabilities based on conditional branches as they use the Pattern History Table (PHT) for target prediction. Spectre-STL exploits the fact that stores may not always become visible to subsequent loads via the Store To Load (STL) buffer. While indirect branches (which enable Spectre-BTB attacks [76]) also occur in BPF, they are not as common and the program cannot use them directly. Because of this, the *retpoline*-based defense against Spectre-BTB is by default always enabled for privileged and unprivileged BPF programs, as it is also the case for the rest of the kernel [23, 78]. We therefore stick to the default and always keep them enabled.

2.1.2 Side Channels. Transient execution attacks can use a variety of shared hardware components to communicate sensitive data to attackers. Notably, this does not only include caches but also microarchitectural execution *ports* with simultaneous multithreading [22]. In this work, we assume the established **constant time (CT) leakage model** to capture all these incidental channels [36] by assuming that all accessed addresses are potentially leaked. This includes both instruction- and data addresses, but not the respective values at these addresses. From this, it follows that one must not branch based on secret data nor use secrets as memory offsets.

2.1.3 Unsafe Information-Flow. The CPU may even load additional sensitive data (erroneously) during speculative execution. For example, *speculative type confusion* [74] can cause the CPU to load sensitive data from an attacker-controller address and subsequently leak the data through a covert channel. To prevent any such unsafe information flow, Speculative Constant-Time (SCT) [35] and Speculative Non-Interference (SNI) [56] are notable formal notions that capture the information-flow properties of the victim program that enable transient execution attacks [36]. They are commonly referred to as *speculative security properties.*

SCT extends the constant-time programming paradigm, which modern cryptographic programs commonly use, to prevent Spectre gadgets by ensuring that the code executed speculatively does not leak sensitive data [35, 36]. A program that satisfies SCT must only operate on sensitive data using CT processor instructions [14, 66, 67]. SCT can be efficiently enforced using type systems that model the sensitive data and its permitted operations [111, 119].

SNI formalizes the intuition that speculation can only leak data, which the program already leaks during architectural execution [36, 44, 56, 100]. For example, if some scalar is already leaked into a side channel architecturally, there is no point in protecting it from speculative leakage. To enforce SNI for a sandboxed program, [56] uses symbolic execution and a satisfiability modulo theories (SMT) solver. Like SCT, SNI not only protects the sandbox runtime (e.g., the kernel) but also the sandboxed program itself (e.g., a BPF program) from Spectre attacks (i.e., they prevent *poisoning attacks*) [37].

While enforcing either SCT or SNI prevents Spectre gadgets, SCT suffers from false positives while SNI cannot be efficiently enforced for arbitrary programs. Applying them to BPF is therefore not necessarily useful. Neither is applying them trivial as the verifier was not developed with SCT or SNI in mind. However, in Section 3, we retrospectively analyze whether the BPF verifier enforces SCT or SNI for any of BPF's data types.

2.2 Spectre Defenses of Linux BPF

In this section, we present the first thorough analysis of BPF's Spectre defenses. We build upon previous talks and reports on the topic [23, 78, 105] but add more details and, in the following section, discuss whether BPF enforces any speculative security properties. We enable future work as BPF's defenses to date are undocumented [5]. Also, the number of contributors is low [27, 28, 86, 114] and both our work [46–48, 50] and others [25, 26] have repeatedly discovered bugs and misleading code.

2.2.1 Whether to Enable Defenses for Privileged BPF. While the need for applying Spectre defenses to unprivileged BPF is clear, this section presents the arguments for deciding whether to also enable defenses for privileged users. As of today, privileged BPF users are

```
I: // reg = is_ptr ? public_ptr : scalar;
A: if (!is_ptr) goto C; // mispredict
B: value = *reg;
   covert_channel[value];
C: exit();
```

Figure 2: Example from a BPF program that contains a speculative type confusion gadget and is rejected by the Linux v6.5 BPF verifier without VeriFence.

only restricted regarding CPU time (bounded loops) and memory usage (memory/type-safety). BPF's Spectre-PHT and Spectre-STL defenses however are kept disabled by default as they may lead to program rejections and are expected to have performance overheads. With small programs, this is acceptable because privileged users can be trusted to not insert Spectre gadgets into the kernel on purpose.

With a rising number of privileged BPF programs being used in production, there is also a rising risk that gadgets from privileged programs are injected into the kernel *by mistake*. The kernel assumes that privileged users manually check their programs for Spectre gadgets to prevent attacks on the BPF program and on the whole kernel. We argue this assumption is unrealistic, as most developers are not familiar with transient execution attacks. Further, when writing BPF programs in high-level languages such as C, Rust, or Python, gadgets are not directly visible and may only occur because of compiler optimizations. Also, BPF programs are frequently generated ad-hoc [127] and thus never undergo code review for Spectre gadgets.

Figure 2 shows one example for a program vulnerable to Spectre-PHT that may be loaded by an admin unfamiliar with transient execution vulnerabilities. This effectively places an arbitrary-read gadget in the kernel that can be used by anyone (even unprivileged or remote users [107]) who controls the invocation of the BPF program, its input parameter `scalar`, and the covert channel used (`covert_channel[value]`).

To date, there exists no reliable static analysis tool that can find actually exploitable Spectre gadgets without false positives. Further, searching for these gadgets manually is extremely time-consuming. It is therefore an open question to which extent exploitable Spectre gadgets exist in real-world privileged BPF programs. Our work allows cautious users to still eliminate this potential attack vector by enabling full Spectre defenses for privileged BPF without risking program rejections.

2.2.2 *Attacker Model.*

We consider both unprivileged attackers that can load arbitrary unprivileged BPF programs into the kernel, and attackers that can coerce a privileged user into loading architecturally-safe privileged BPF programs into the kernel on their behalf. As of Linux v6.5, the kernel can only effectively protect itself from the former.

We include the second type of attacker because many privileged programs are not sufficiently reviewed for Spectre gadgets or generated ad-hoc [127]. They may therefore unintentionally contain exploitable Spectre gadgets as we have discussed in the previous section.

2.2.3 *Architectural Safety.*

To understand the existing Spectre defenses, which ensure speculative safety, we first briefly present the design of the verifier and its mechanisms to ensure architectural safety. The main goal of the verifier is to limit the damage malicious or buggy BPF programs can cause. For this, it verifies a bounded execution time and memory safety but not functional correctness. Bounded execution time is mainly useful for allowing BPF use in interrupt contexts, while memory safety prevents memory leaks and program bugs that would easily enable kernel exploits.

To restrict data flow into the BPF program's registers and stack, the verifier enumerates all possible paths through the BPF program and simulates each path's execution. To verify memory and type safety, the kernel analyzes the types (mainly different classes of pointers and scalars [115]) and the value ranges of the scalars (for which it uses *tristate numbers* [121]). This allows the verifier to ensure that all scalars used as pointer-offsets only point to locations owned-by or borrowed-to[1] the BPF program. For example, this prevents OOB accesses to the BPF stack and to network packets (the BPF program can manipulate packets directly using its context pointer [46]). Further, accessing uninitialized stack slots and registers is also prohibited. In summary, the BPF program can only access memory locations (i.e., load their value into registers) to which the kernel grants explicit access.

The kernel not only restricts how data flows into BPF program registers and stack but also limits how the program uses the data in these locations. For this, the verifier distinguishes between pointers and scalars. This allows the verifier to limit how the BPF program processes data and ensure that only safe operations are executed. For example, programs can only dereference pointers (at valid offsets) and not cast them into scalars. Further, they can only use scalars in arbitrary arithmetic logic unit (ALU) operations and conditional branching based on their value [46]. Therefore, the BPF program can only use kernel pointers in CT operations (except for dereferencing them), while scalars are only restricted so that the program cannot cast them into pointers.

2.2.4 *Design Goals.*

By extending the design used to ensure architectural safety, the kernel's Spectre defenses for BPF also enforce that the same restrictions apply in speculative execution as well. The goal in Linux v6.5 is only to protect the kernel from BPF programs and associated user applications. The BPF program itself is not protected against other user applications that might exploit Spectre gadgets to retrieve scalars (including cryptographic keys) the program processes. Further, the existing defenses against Spectre-PHT and Spectre-STL try to be transparent to users as much as possible, as the kernel applies them to the bytecode during verification time. The kernel does not require the user or source compiler to insert speculation barriers into the program. However, the defenses fail to be entirely transparent, as they can impact performance and prevent programs from passing verification.

2.2.5 *Spectre-STL.*

To defend against Spectre-STL (v4), the kernel inserts speculation barriers after *critical* stores to the BPF stack [27]. A store is *critical* if speculatively bypassing it would make otherwise inaccessible data (or operations upon data) available to the program.

---

[1] For example, this includes pointers from the `bpf_ringbuf_reserve()` helper which must be either submitted or discarded subsequently [4].

For example, initializing a stack slot or overwriting a scalar value with a pointer are critical stores as the kernel prohibits reading uninitialized stack slots and dereferencing scalars. The verifier can only skip the insertion of a barrier if a scalar is overwritten with another scalar [50]. Confusing one scalar value with another scalar cannot lead to OOB memory accesses since the verifier enforces pointer limits using branchless logic (e.g., masking). This is further discussed in the following section, as it is also required to defend against Spectre-PHT.

This defense is expected to have an impact on BPF's performance as the verifier inserts speculation barriers that reduce instruction-level parallelism (even if there is no misprediction). However, this impact has not been measured exhaustively for real-world applications, and we therefore include it in our evaluation.

2.2.6 Spectre-PHT. While Spectre-STL effectively causes the CPU to speculatively bypass a store, Spectre-PHT (v1) causes the CPU to bypass evaluation of a branch condition. Therefore, conditional branches can no longer be reliably used to ensure memory and type safety. To defend against this, the verifier prevents OOB accesses using branchless logic [25, 86, 114] and type confusion by verifying architecturally-impossible speculative execution paths [28].

Branchless Bounds Enforcement. This includes simple masking but also more complex instruction sequences when required. First, for arrays (i.e., BPF maps) the kernel can simply ceil the size to a power of two and apply the respective index-mask before the access [114]. Second, for pointers, the kernel deducts the bounds from the conditional branches that lead to the pointer dereference. Then the kernel also enforces them directly before the access using a special sequence of ALU operations [25] (additionally, the verifier has to check a remaining corner case using verification of an architecturally impossible speculative code path). Third, for the stack, the kernel enforces that all offsets are constant to simplify the implementation [86]. Finally, there is one exception where the kernel allows the program to access OOB memory speculatively for practical reasons [113]. We have verified that the specific memory layout in this case does not expose any secrets [47].

Verification of Speculative Execution Paths. While branchless bounds enforcement prevents the BPF program from accessing forbidden data, it cannot prevent the program from using kernel pointers or scalars in an unsafe manner (e.g., dereferencing a scalar). To ensure speculative type safety, the kernel also simulates and checks the execution paths that include mispredicted branches [28]. The verifier only allows transient behavior that is also permitted architecturally. Even though this is a sound approach, it suffers from false positives because not every architecturally unsafe operation enables a Spectre attack [56] (and the architectural verification logic itself also already suffers from false positives). Using separate simulation and verification logic for the transient domain could resolve this, but it would increase the verifier's attack surface even further.

In summary, branchless bounds enforcement and verification of mispredicted branches enable reliable defense against Spectre-PHT.

3 SECURITY ANALYSIS

In this section, we analyze the foundations of BPF's Spectre defenses. We do this both from a hardware and a formal perspective. Regarding the hardware, we summarize the required instruction-set properties for the defenses (i.e., the hardware-software contract). Further, we retrospectively analyze the speculative security properties the verifier enforces.

3.1 Hardware-Software Contract

To terminate speculative execution after an STL misprediction, the verifier relies on speculation barriers. For this, it uses the undocumented `nospec` BPF bytecode instruction that is not available to user space. Notably, on x86-64, the just-in-time (JIT) implements these using the `lfence` instruction, which is in line with Intel's recommendation for Spectre-STL and -PHT [63, 64]. Independent research has confirmed that `lfence` terminates speculation [85], therefore, we deem them reliable. On ARM64, the JIT compiler lowers `nospec` to a no-op because the hardware already defends against Spectre-STL in firmware [13]. To perform safe pointer arithmetic, the verifier further relies on ALU instructions that have data-independent timing [14, 66].

3.2 Speculative Security Properties

The verifier enforces a mix of speculative security properties for BPF programs. Overall, the verifier attempts to prevent the BPF program from speculatively breaking out of its sandbox and is successful in that (to the best of our knowledge), except for the exception discussed in Section 2.2.6 [47]. *Speculative-breakout attacks* are therefore prevented. Regarding the exception, we are also unable to construct an exploit that leaks *sensitive* kernel data [47]. However, while this in summary protects the kernel, the verifier does not enforce SNI [36] as processed scalars, which are never leaked architecturally, can still leak after a speculative scalar confusion due to Spectre-STL. Therefore, there is no support for BPF programs in protecting cryptographic keys they process. While users cannot insert speculation barriers manually in Linux v6.5, the kernel could offer basic support for this by exposing the internal BPF bytecode instruction to insert barriers (`nospec`) to users. Regarding SCT, the verifier enforces it only for pointers, but with the exception that the program can dereference the pointer itself. Overall, retrofitting a single formal speculative security property to BPF does not appear viable. However, we still find SNI and SCT to be of high value as we discover multiple kernel bugs by analyzing BPF's defenses with them in mind [46, 48, 50].

4 PROBLEM STATEMENT

Out of the limitations we have identified throughout the previous sections, we deem the Spectre-PHT defenses leading to program rejections to be the most limiting to users: If the verifier finds a speculative execution path (following a simulated misprediction) that performs prohibited operations, it rejects the whole program [28]. This forces users to resort to tedious restructuring of the source code or abandon BPF completely (which has a much higher performance impact than the speculation barriers [49]). Frequently, this development overhead motivates users to simply disable defenses altogether [109], which enables BPF-based exploits as shown in the original Spectre paper [76].

With Spectre-PHT identified as the most limiting problem, we focus on the respective rejections in this work and leave solving the lack of support for SNI (Section 3.2) and the performance regressions due to Spectre-STL defenses (Section 2.2.5) to future work.

5 DOMAIN ANALYSIS

As discussed in the previous section, Linux v6.5's Spectre-PHT defenses cause BPF program rejections when the verifier cannot prove some transient execution is safe. In this case it prevents unprivileged users from using BPF altogether. At the same time, the Spectre-STL defenses can negatively impact the execution time as they insert speculation barriers. However, to-date it is unclear to which extent these defenses are actually triggered in real-world BPF programs.

In this section, we present a domain analysis of the BPF program landscape to analyze how BPF programs are affected by the kernel's Spectre defenses. We collect 364 BPF object files containing 844 individual BPF programs from six popular open-source projects and analyze the number of objects rejected and speculation barriers inserted. To support future research, we publish our tools for collecting and analyzing the programs.[2]

5.1 Dataset

We include a diverse set of programs taken from the Linux kernel selftests and BPF samples [77], libbpf examples [89], Prevail [52], BCC [112], the Cilium Kubernetes Container Networking Interface (CNI) [116], the Parca Continuous Profiler [68], and the Loxilb Network Loadbalancer [99][3]. We observe that programs from the Linux kernel selftests are often designed to only test a specific kernel interface and are therefore very small. This motivates us to further group the programs into **test or example** and **application** programs (171 programs from 50 object files). The application group only includes programs from the BCC, Loxilb, Parca, and Cilium projects, as well as 2 specific object files from the Linux selftests with programs adapted from real-world applications.

5.1.1 *Applicability to Unprivileged BPF.*

Unfortunately, because of their high risk for rejection by the verifier, few real-world BPF programs designed for unprivileged use exist today. We aim to enable more real-world BPF programs by reducing unnecessary program rejections. To still ensure a reasonably-sized dataset for our evaluation, we modify the kernel to support activation of Spectre-PHT and -STL defenses at runtime and then also enable these defenses for existing privileged BPF programs we collect from the open source projects.

Still, our evaluation accurately represents the real-world unprivileged BPF programs that will be developed for our motivating use-cases from Section 1 in the future. This is because, at a bytecode-level, the program behaviors that lead to Spectre induced program rejections or performance-regressions are the same for unprivileged and privileged BPF, as they primarily differ in the kernel interfaces (i.e., attachment point and BPF helper function calls) they interact with. The compiler toolchain, kernel infrastructure, and high-level verification algorithm are exactly identical for privileged and unprivileged BPF. Programs loaded for privileged and unprivileged use only differ in that the verifier ignores minor unsafe behaviors for privileged programs (basically assuming they must be false positives as the user is trusted) while triggering a rejection for programs from unprivileged users.

Upstream kernel developers have also used this technique to analyze the impact of the Spectre-PHT defenses on pointer arithmetic [24]. We have validated the maintainer's reasoning here and see no technical reason why the conclusions drawn from our dataset should not apply to future unprivileged BPF programs.

5.1.2 *Program Size.*

To inform our analysis, we first measure the number of BPF bytecode instructions per program. We observe that the median number of instructions per program is only 40 overall. For the programs classified as applications, the median is 46 instructions per program, while the arithmetic mean is 559 since there are some programs that are close to the verifier's complexity limit of 1×10^6 instructions (e.g., Parca's stack sampler). Overall, the low median number of instructions is expected as most BPF programs only implement a fast path or policy decision in the kernel.

5.2 Speculation Barriers

We first analyze how many speculation barriers the verifier inserts to defend against Spectre-STL. Figure 3 shows the percentage of barriers per BPF program for all collected object files that are compatible with our kernel version. Comparing the programs classified as test or example programs with those from application projects, we find that application programs require more barriers per instruction on average (2.2 % instead of 1.0 % overall). This is likely because they are usually more complex and thus cannot work with registers exclusively. Simply counting the number of speculation barriers, of course, only indirectly relates to a real-world performance overhead, which heavily depends on the exact location of the barrier (e.g., in a tight loop or only at the beginning of the program during initialization). In Section 8, we will, therefore, analyze the performance overhead of the Spectre-STL defenses in real-world execution-time benchmarks. Still, we expect the overall overhead to be much lower compared to the performance overhead if BPF were not used at all [49].

5.3 Program Rejections

While the Spectre-STL defenses of Linux v6.5 only slow down the BPF program, the Spectre-PHT defenses can lead to the entire program being rejected. This forces users to resort to tedious restructuring of the source code or abandon BPF completely. In this section, we analyze how many programs from real-world projects the verifier actually rejects.

Figure 1 from Section 1 shows the number and cause of program rejections with only Spectre-BTB defenses active (*BTB*), Spectre-STL and -BTB defenses active (*STL+BTB*), and with additional Spectre-PHT defenses active (*Full*, i.e., PHT+STL+BTB). As expected, the Spectre-STL defenses cause no rejections. However, with the Spectre-PHT defenses active, the verifier rejects 31 % of the 364 compatible BPF object files we have collected. Excluding test or example programs, we even find that 54 % of the remaining 50 application object files are rejected (likely because they are usually more complex and, therefore, more likely to contain any

---

[2] https://gitlab.cs.fau.de/un65esoq/bpf-spectre

[3] Despite our best efforts, we are unable to use most of the BPF objects from the Prevail paper because they are not compatible with Linux v6.5. For the Cilium project, only the `bpf_sock` program is compatible with our `bpftool`-based toolchain [118].

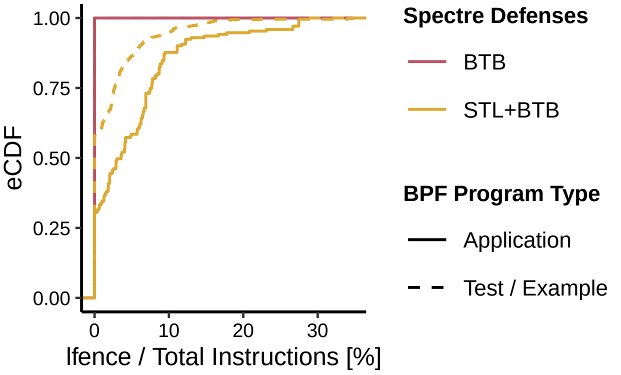

Figure 3: Percentage of speculation barriers in 844 BPF programs as a fraction of the total number of machine instructions with Spectre-STL defenses active.

unsafe speculative behavior). We group the causes for rejections into four categories:

- **Types** (15 %)**:** These are type errors on speculative program paths. For example, speculatively casting as scalar into a pointer and dereferencing it.
- **Variable Stack Access** (7 %)**:** The program contains a speculative or architectural stack access for which the verifier cannot statically compute the offset. To simplify the Spectre-PHT defenses, kernel developers have disallowed variable-stack accesses [86].
- **Breakout** (5 %)**:** Unsafe speculative execution where the BPF program accesses locations (memory or uninitialized registers) owned by the kernel.
- **Too Complex** (4 %)**:** Because the verifier has to check the additional speculative paths, some programs exceed the verifier's complexity limits (e.g., a maximum path length of 1×10^6 instructions).

In summary, the verifier rejects BPF programs due to a diverse set of unsafe behaviors that can lead to transient execution attacks on the kernel if the program leaks the resulting secret into a side channel subsequently.

In Table 1, we further analyze the number of rejections per software project. As expected, there is no notable difference in the extent to which the rejections affect the different projects since most of the bytecode-level properties that lead to unsafe speculation do not directly map to high-level constructs in the source code. The only exceptions are architectural variable-stack accesses [86], which users can avoid by not allocating arrays on the stack. In summary, all but one of the projects we have analyzed (that is Cilium, for which we only have one program compatible with our toolchain) are negatively affected by the Spectre-PHT defenses.

To conclude, we find that the Spectre-PHT defenses in Linux v6.5 are significantly more limiting to users than the Spectre-STL defenses, showcasing the significance of our work. The defenses cause users to resort to disabling defenses altogether (e.g., [109]), which opens the door to introducing dangerous arbitrary-read gadgets into the kernel. In the following section, we present VeriFence, which defends against Spectre-PHT without rejecting the whole BPF programs.

Table 1: Number of BPF object files rejected with the existing Linux v6.5 defenses for different software projects.

Project	# Programs	# Files	# Files Rejected
Linux Selftests	592	275	80
BCC	133	39	19
Linux Samples	71	32	5
Loxilb	19	4	3
Cilium	10	1	0
libbpf Examples	10	7	1
Parca	7	4	3
Prevail	2	2	1

6 DESIGN

As outlined in the previous sections, the current BPF verifier pessimistically rejects programs that pose a potential security threat. While this is preferable to leaving the system vulnerable to transient execution attacks, it also limits BPF's usability. These rejections seem unnecessary, especially considering that effective defense mechanisms exist, which could be included in the program instead.

In this section, we present the idea and design of **VeriFence**, not only a solution to the problem from Section 4, but also a generic technique to build Spectre-resistant software sandboxes based on verification. VeriFence optimistically attempts to verify all speculative execution paths and only falls back to speculation barriers when unsafe behavior is detected. Importantly, we fully reuse existing verification logic for this to balance verifier complexity (and thereby potential for kernel bugs) with BPF execution-time overheads. Surprisingly, our evaluation shows that this approach results in low application overheads because invocation-latency is more critical to BPF's performance than code execution-time. While we only implement VeriFence's approach for Spectre-PHT, it can also be used to reduce the number of speculation barriers required for Spectre-STL defenses [27, 50], and to Spectre-BTB with Intel's Control Flow Enforcement Technology (CET) [108]. We first present our idea by example and then discuss its security and performance.

6.1 Fence or Verify

Based on our analysis of the BPF program landscape, we have identified the existing Spectre-PHT defenses to be the main real-world issue because they prevent applications from using BPF altogether. In this section, we analyze a minimal example, shown in Figure 2, of a BPF program where VeriFence successfully prevents the critical transient execution while the Linux v6.5 BPF verifier rejects the entire program [28]. The program executes either with `is_ptr` true or false, and we assume logic in block `I` implements `reg = is_ptr ? public_ptr : scalar`. We exclude the corresponding branch in block `I` in this analysis to give a concise example. Figure 4 illustrates all execution paths through the BPF program. Path 1 and Path 2 are architecturally possible paths through the program. By simulating the execution of each basic block, the verifier computes the invariants that will hold after executing the block based on the inputs,

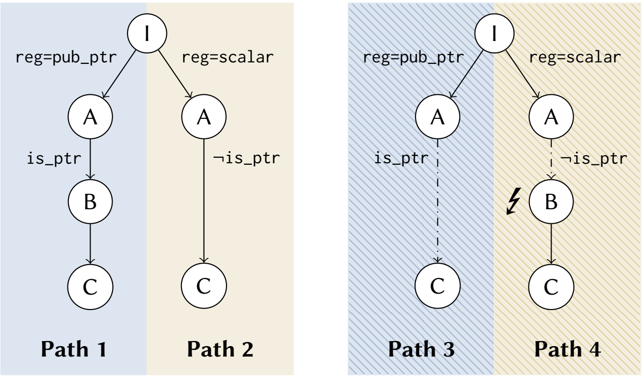

Figure 4: Execution paths through the BPF program from Figure 2. Paths marked with → denote architectural execution, while a dashed arrow indicates speculative execution. ↯ indicates unsafe behavior the verifier must prevent.

the block's instructions, and the condition of the final branch that ends the basic block (e.g., `reg` will be equal to `public_ptr` and `is_ptr` must be true, when we go to block `B` after `A`). We list these invariants on the edges connecting the basic blocks in Figure 4.

Only considering the architectural execution paths, the program could be accepted because the verifier understands that block `B` will only execute if `reg` contains a valid pointer (in Path 1). Thus, the program does not exhibit any architecturally unsafe behavior and was therefore accepted by the kernel before 2018. However, with the defenses active (specifically [28]), the program is rejected as the verifier also simulates branch mispredictions, which leads to the speculative execution paths (Path 3 and 4) shown in Figure 4. With Spectre defenses, the verifier rightly finds the program to be unsafe because it performs a forbidden pointer dereference on Path 4. Specifically, when block `B` executes with a scalar in `reg`, the program breaks out of it sandbox by accessing an arbitrary address (↯ in Figure 4). Subsequently, the resulting kernel secret could be leaked into a covert channel (denoted as `covert_channel[value]` in the code example). In the Linux v6.5 kernel, this unsafe behavior causes the whole BPF program to be rejected, which either prevents unprivileged users from using the program altogether or forces privileged users to disable the Spectre defenses.

Our solution, VeriFence, solves this by dynamically falling back to inserting a speculation barrier whenever the verifier detects unsafe program behavior after a simulated misprediction. By utilizing speculation barriers instead of program rejection to protect against unsafe program behavior, VeriFence successfully addresses Section 4. Further, whenever the verifier encounters a barrier while verifying a speculative path, it can stop verifying this path. Both of these modifications are based upon the insight that speculation barriers behave like program exits when the execution is transient. In the example from Figure 2, VeriFence triggers the insertion of a barrier at the beginning of block `B` before `reg` is dereferenced. This still allows the CPU to mispredict the branch but terminates the speculative execution path when block `B` is reached. In contrast, the speculative execution in Path 3 is not problematic because no unsafe operation is performed in block `C`, even though the CPU mispredicted the branch at the end of block `A`. VeriFence rightly detects this and does not insert a speculation barrier at the beginning of block `C`.

6.2 Security

VeriFence only allows transient behavior that was already allowed architecturally and prevents all other execution paths using speculation barriers. We can therefore reduce its security to the security of the existing BPF verifier. Importantly, this allows VeriFence to benefit from formal verification efforts that improve the security of the verifier [92, 121], even if they do not explicitly take Spectre into account. In summary, VeriFence uses a reliable and easy-to-implement method to prevent unsafe speculative behavior (and thereby the use of secret-leaking side- and covert channels).

Our extension does not change the speculative security properties enforced by the BPF verifier, because we do not permit any new speculative behavior. With the assumptions noted in Section 3, our modified verifier is therefore still *secure*, assuming the CT leakage model, an unprivileged local attacker, and the secrecy policy identified in Section 2.2.3.

While VeriFence does not affect security against unprivileged attackers, it improves security in the light of privileged users that only manually verify architectural safety. With VeriFence, these users no longer accidentally load BPF programs into the kernel that unexpectedly leak sensitive kernel or user data. For example, loading the program from Figure 2 into the kernel effectively places an arbitrary-read gadget in the kernel that can be used by anyone (even unprivileged or remote users [107]) who controls the invocation of the BPF program, its input parameter `scalar`, and the covert channel used (`covert_channel[value]`). With VeriFence enabled for privileged users (which becomes practical because of the improved expressiveness), no arbitrary data will be read into `value` and therefore the kernel remains secure.

In summary, VeriFence soundly prevents unsafe transient execution and directly benefits from work that discovers architectural verifier bugs. The main expected downside of VeriFence's approach, in comparison to a vulnerable system, is the execution-time overhead at runtime due to the speculation barriers and the added verification-time overhead. Both are the topic of the following section and are further analyzed in the evaluation.

6.3 Performance

In this section, we analyze VeriFence's impact on performance. In particular, VeriFence has an impact on the execution time of the BPF program and on the verification time when loading a BPF program.

VeriFence inserts speculation barriers into the BPF program, which reduce the instruction-level parallelism inside the BPF program. However, the user and kernel code that calls the BPF programs, as well as the kernel helper functions invoked by the BPF programs, are unaffected by this change and therefore still execute at maximum performance. As most applications only spend a small part of the CPU time executing BPF code, we expect the real-world overhead to be small, which is also supported by our evaluation. In any case, VeriFence enables unprivileged users to use BPF at all, as their programs were rejected prior to VeriFence. This saves them from having to implement their logic in user space, which would then require very expensive user/kernel switches [49]. We therefore deem VeriFence's negative impact on BPF's performance the more favorable trade-off.

Further, the number of barriers inserted by VeriFence is reduced in comparison to more naive approaches, as barriers are only inserted when the verifier actually detects unsafe behavior that could enable a transient execution attack on the kernel. In addition, verification of a speculative path is cut short when a barrier is already present. This not only reduces the verification time but also helps to keep the number of barriers inserted small. At the same time, this approach avoids complex compile-time analysis, which would not be practical for BPF. Our evaluation supports that this approach is precise even though it is simple – we find that VeriFence inserts a lot fewer barriers than the Spectre-STL defenses already present in Linux v6.5.

Aside from the execution-time overheads, VeriFence only impacts the verification time (in comparison to a vulnerable system) for the programs that would otherwise have been rejected. For these programs, VeriFence continues to explore the remaining architectural and speculative paths after discovering unsafe behavior. We do not make any assumptions about the size of the CPU's speculation window, following the reasoning from [36]. However, if the hardware were to expose this information reliably, VeriFence could use it to significantly reduce the verification time because the verifier indeed knows the exact microarchitecture on which the code will run. To limit verification time in our prototype, we selectively cut explored speculative paths short by inserting a barrier prematurely when we approach the verifier's configurable complexity limits (e.g., number of instructions simulated, number of branches followed[4]). Using this simple heuristic, we can even verify the largest BPF application programs accepted by the Linux v6.5 verifier without defenses. In summary, there are multiple approaches that would allow us to reduce the number of barriers even further without impacting verification complexity. Because verification is usually not part of the user application's hot path [39] we focus on the execution-time overhead in our evaluation.

In summary, VeriFence improves the performance of unprivileged user applications by allowing them to use BPF at all. At the same time, our approach has no impact on privileged users, who can still dynamically disable VeriFence per-program at execution time if they choose to manually check for gadgets.

7 IMPLEMENTATION

We implement VeriFence for the Linux [77] BPF verifier and publish our patches[5] consisting of less than 1000 source line changes under an open-source license. The majority of changes merely restructure the existing BPF verifier to support our design. While we use Linux v6.5, our approach is not fixed to only apply to this specific Linux version. Instead, it even benefits from future improvements to the precision of the verifier's architectural analysis (which also reduce false-positives for VeriFence) and JIT compiler. From a practical perspective, we are not aware of any fundamental reasons for which the patches could not be merged into upstream Linux.

To make our approach portable, we introduce a distinction between speculation barriers (verifier-internal BPF bytecode instructions) against Spectre-STL, and BPF speculation barriers against Spectre-PHT into the kernel (`nospec_v4` and `nospec_v1` respectively). The verifier inserts both barriers into the bytecode, and the JIT compiler backends then either drop or lower the instructions based on the architecture's configuration. For example, ARM64 does not require speculation barriers to defend against Spectre-STL due to its firmware defenses [13] while x86-64 does require `lfence` instructions for both Spectre-STL (unless Speculative Store Bypass Disable, SSBD, is active) and Spectre-PHT. This approach is also compatible with research that proposes address-specific speculation barriers like `protect` from [119]. In summary, vulnerability- and even address-specific speculation barriers allow the verifier to remain architecture-agnostic while not impacting performance on architectures that are not affected.

8 EVALUATION

We evaluate VeriFence using both static analysis and real-world execution time benchmarks. Regarding the former, we analyze whether VeriFence successfully protects the programs rejected by the upstream Linux v6.5 verifier and how many speculation barriers it inserts. Regarding the latter, we analyze the performance impact of the speculation barriers on three popular performance-critical applications of BPF. As discussed in Section 5.1.1, our results carry over to future unprivileged BPF programs even if most of the real-world BPF programs used today (and therefore also in our evaluation) still require root privileges.

8.1 Static Analysis

First, we apply VeriFence to all BPF programs from our motivating analysis of the BPF program landscape from Section 5. VeriFence successfully applies defenses to all real-world application programs in our dataset and the number of barriers it inserts is insignificant in comparison to the upstream Spectre-STL defense.

8.1.1 Program Rejections. Figure 5 compares the number and type of verification errors with VeriFence to the upstream Linux defenses (in both cases the Spectre-STL defenses are also active). As expected, VeriFence is able to successfully protect all BPF programs from our application group and almost all programs from the full dataset. The remaining programs are all from the Linux kernel selftests which VeriFence rejects for one of two reasons:

- **Variable Stack-Access (5/364):** The verifier currently does not implement defenses against Spectre-PHT in the light of variable stack accesses to simplify the implementation [86], our modified verifier therefore still rejects them. We deem an extension to resolve this out of scope as one can easily avoid variable stack accesses in real-world BPF programs. Our evaluation confirms that rejection due to variable stack accesses is not a problem for any of the application's BPF programs.
- **Too Complex (10/364):** The Linux kernel selftests contain object files with very large programs to test the verifier's complexity limits. Because VeriFence has to verify additional speculative program paths, verification can fail if the existing program was already designed to be as large as possible. If

---

[4] For our benchmarks, we increase the existing limits by a factor of 4 to successfully verify all real-world BPF programs (designed for verification without Spectre) with full defenses. For Parca, we increase the limit by a factor of 32.

[5] https://gitlab.cs.fau.de/un65esoq/linux/-/tree/bpf-spectre

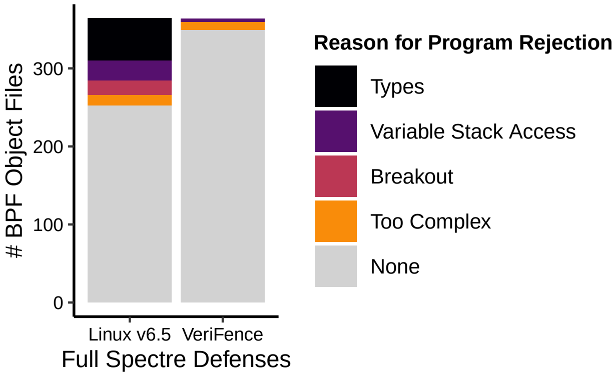

Figure 5: VeriFence successfully applies to all real-world application programs without modifications. Analyzing all 364 object files, defenses only fail to apply to 4 % which is a significant improvement over the upstream verifier's 31 %. VeriFence only rejects 15 test programs from the Linux selftests which exhibit unverifiable architectural behavior that could easily be avoided in real applications.

> users were to encounter this in the real-world, they could easily circumvent it by splitting their program into multiple smaller programs that call each other using BPF tail calls.

In summary, **VeriFence successfully verifies all BPF programs from real applications, thereby solving the problem described in Section 4.** Further, the remaining theoretical reasons for failed verification are easy to understand and circumvent by BPF program developers.

8.1.2 Speculation Barriers. While it is VeriFence's desired outcome to enable more programs to be successfully loaded into the kernel, it also has the potential downside of slowing down those programs because of the speculation barriers it inserts for defense. In this section and the following real-world performance evaluation, we analyze the extent to which these barriers affect performance.

First, we repeat the analysis from Section 5.2 for VeriFence, analyzing the number of speculation barriers it inserts in comparison to the Spectre-STL defenses. Figure 6 shows the fraction of BPF programs with less than X percent of speculation barriers for VeriFence and Spectre-STL defenses (excluding the 15 rejected programs from the previous section). VeriFence only inserts barriers when the program would otherwise have been rejected by the verifier, therefore the overall number of barriers it inserts is small.

In summary, we find that VeriFence has a low expected performance overhead for a diverse set of applications. In the following section, we will back up this claim using performance benchmarks for three real-world applications using BPF.

8.2 Real-World Performance Evaluation

In this section, we analyze the real-world application overhead of VeriFence for over 50 distinct BPF programs. We focus on event tracing, continuous profiling, and packet processing, as these are three of the most popular performance-critical applications of BPF.

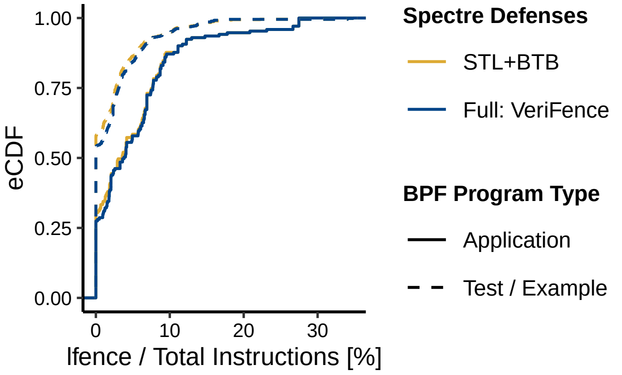

Figure 6: VeriFence inserts only insignificantly more speculation barriers than the Spectre-STL defenses present in the upstream Linux kernel.

As more complex programs are more likely to exhibit unsafe transient behavior, we select the Parca Continuous Profiler [68] and the Loxilb Network Load Balancer [99] to analyze VeriFence's overhead in extreme cases.

We run all our execution-time benchmarks in a Debian 11 GNU/Linux system with a v6.5.11 kernel on a 6-core 2.8 GHz Intel CPU from 2017 (Intel Core i5-8400). We disable dynamic voltage and frequency scaling (DVFS) as the execution-time measurements would otherwise be highly dependent on the current system load [125], which is not relevant to our evaluation and therefore only hinders reproducibility. In our graphs, we use standard boxplots [124] showing the first, second (i.e., median), and third quartile.

8.2.1 Event Tracing. Tracing is one of the most popular applications of BPF useable both for performance debugging and continuous monitoring in production. It allows users to gain valuable insights into kernel execution with little to no impact on production workloads. Still, their use poses security risks as, without Spectre defenses, BPF programs can easily introduce arbitrary-read gadgets into the kernel by mistake, therefore jeopardizing the security of the whole system. In this section, we analyze the execution-time overhead VeriFence has for BCC's `libbpf`-based tracers when monitoring a system that runs Memcached [79] together with the `memtier_benchmark` [53] load generator. Client and server each use three threads and communicate using Memcached's binary protocol. We perform 15 000 requests taking 18 s on average and repeat each test 100 times.

We analyze 43 BCC tracers[6]. Out of these tracers, 21 require VeriFence to be successfully used with Spectre-PHT defenses. When running the tracers alongside the workload, we find that 22 of the tracers record at least 10 events per second and 9 at least 1000. We conclude that most of the tracers are invoked frequently and collect a reasonable amount of information in this setup. However, the CPU time the tracers spend executing the BPF programs themselves is small. Out of the analyzed tracers, 9 tracers spend more than

---

[6] This includes the 39 BPF objects from Section 5 but the number is higher here because some BPF objects are not designed to be loaded using `bpftool`.

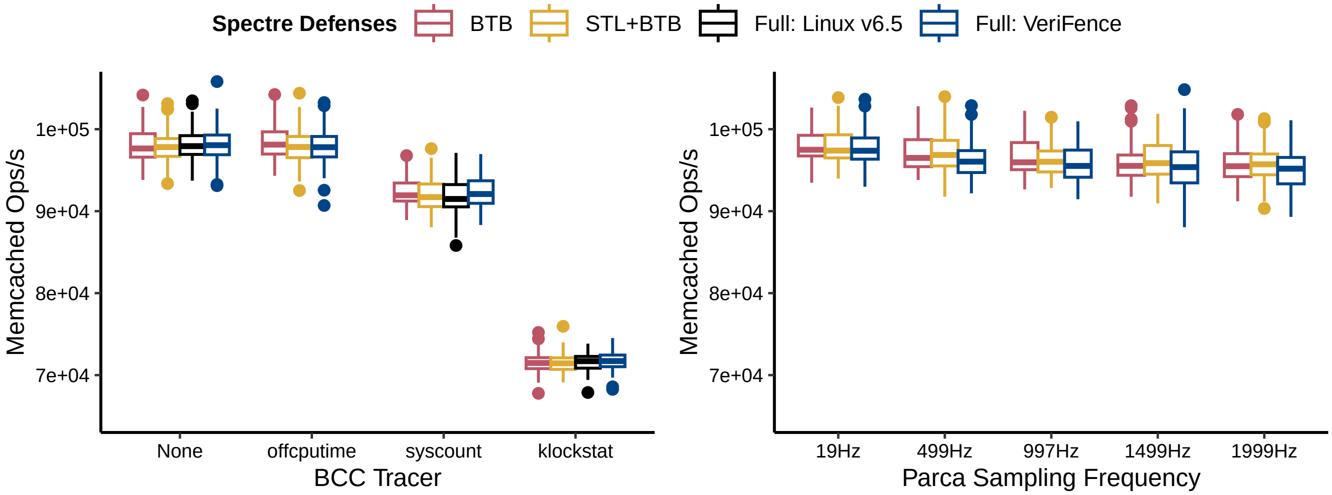

Figure 7: VeriFence does not affect Memcached performance when using any of the 43 analyzed BCC tracers (of which the three most CPU-intensive are displayed in the left plot) or the Parca Continuous Profiler (right plot) to monitor the workload.

0.1 %, and 3 tracers spend more than 1 % of CPU time executing BPF code.

The left plot of Figure 7 shows the impact that the three most CPU-intensive tracers have on Memcached's performance in four different system configurations. Without VeriFence, the verifier cannot successfully apply Spectre defenses to `offcputime`'s BPF program, thus the baseline system configuration (*Full: Linux v6.5*) is missing for this tracer. Defenses do successfully apply to `syscount` and `klockstat` without VeriFence, but we still measure the overhead with VeriFence for completeness. Overall, VeriFence does not have any measurable impact. We also analyze the CPU time spent executing BPF code and find no measurable difference between VeriFence and the other configurations even when we run each test for a total duration of 30 min. This is expected as tracers are designed to not impact production workloads by consuming as few resources as possible. They can therefore benefit from the security benefits VeriFence offers without experiencing increased overheads.

8.2.2 *Continuous Profiling.* The second real-world BPF application we analyze is continuous profiling. Here, a BPF program that records the current user and kernel stack trace is invoked by a timer interrupt at a configurable frequency. The data can later be analyzed to find performance bugs, for example using flame graphs [55]. Traces are either collected continuously in production (sampling frequencies below 150 Hz) or on-demand by developers (usually 500 Hz to 2000 Hz).

For this benchmark, we run the Parca Continuous Profiler alongside the Memcached workload from the previous section. The results are displayed in Figure 7 on the right. As the BPF program that Parca uses to collect the stack samples is relatively complex, it cannot be successfully defended against Spectre-PHT without VeriFence. Therefore, the baseline (*Full: Linux v6.5*) is missing from the figure. Analyzing the amount of CPU time spent in BPF code for this benchmark, we find that it is only 1.0 % even when the maximum sampling frequency we deem reasonable is used. The main overhead of the Parca tracer therefore originates from the code that post-processes the samples in user space. From this, it follows that both the Spectre-STL defenses and the Spectre-PHT defenses using VeriFence do not impact Memcached's performance even though they increase the execution time of the BPF programs by 16 % and 62 % respectively. In comparison to a vulnerable system, VeriFence only reduces Memcached's throughput by 0.8 % at 1999 Hz. At frequencies below 150 Hz, the overhead is no longer measurable (0.1 % for 19 Hz in this particular run). In summary, VeriFence can be applied to Parca's BPF program without overheads to the workload.

8.2.3 *Network Load Balancing.* High-performance packet processing is the application BPF was originally developed for and is still one of the most popular use cases [38, 40, 54, 62, 99, 120, 129].

In this benchmark, we replicate a real-world scenario where multiple Docker containers communicate with each other through the BPF-based Loxilb network load balancer [99]. To stress the load balancer, we extend the benchmarks the Loxilb upstream project includes. To achieve the maximum possible CPU utilization, we do not run the containers on different physical machines. Therefore, the overhead we measure represents the upper bound as the performance is not limited by the speed of the networking interface but only by the CPU's speed.

The four left plots of Figure 8 show the maximum throughput and request rate for Transmission Control Protocol (TCP) and Stream Control Transmission Protocol (SCTP). We normalize each scale to display between 80 % and 105 % of the respective performance achievable with only the Spectre-BTB defenses active (the default in Linux v6.5). For `netperf` [123] and `iperf3` [83], we use one client thread but confirm that multiple threads do not impact our conclusions in separate experiments (e.g., using `iperf` v2.0.14a). We run each benchmark for 2 min and repeat the measurement 10 times.

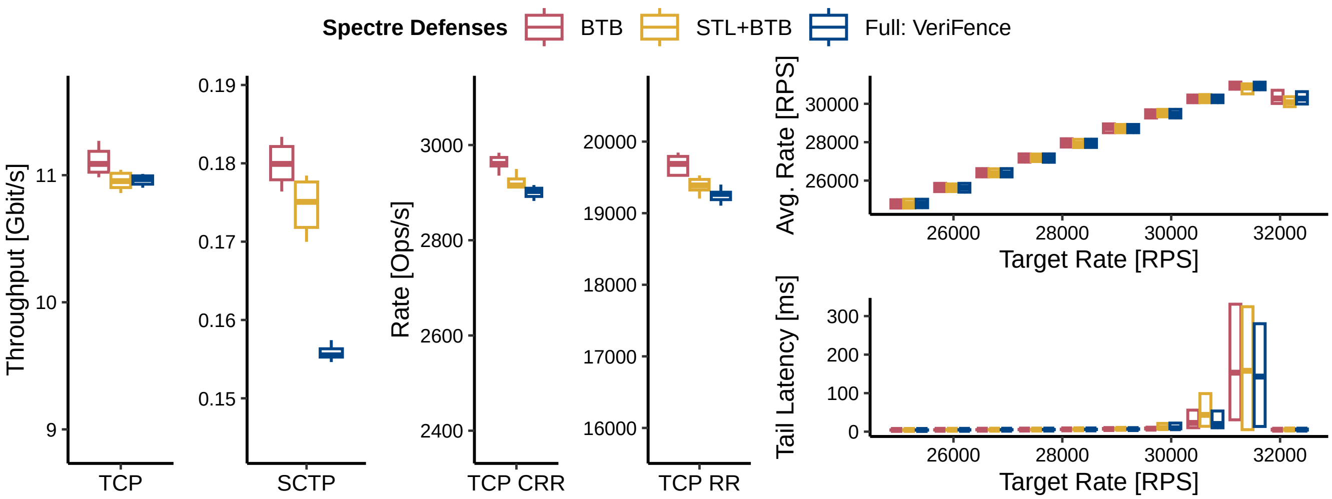

Figure 8: TCP and SCTP throughput, TCP Request/Response- (RR) and Connect/Request/Response (CRR) rate, and `nginx` HTTP tail latency achievable over a BPF-based Loxilb load balancer. The upstream Linux v6.5 verifier cannot successfully apply Spectre defenses to Loxilb's BPF program, therefore the baseline (*Full: Linux v6.5*) is missing.

We observe that VeriFence only has a small impact on TCP throughput and average latency. Compared to a vulnerable system, both VeriFence and *STL+BTB* defenses reduce the TCP throughput by 1.1 % to 1.2 %. As expected, the Request/Response (RR) rate is affected the most by VeriFence with 2.1 % overhead while the CRR rate only decreases by 1.8 %. In summary, the main overhead stems from the Spectre-STL defenses, while VeriFence only has a negligible impact.

SCTP is a message-based protocol that still ensures reliable transportation. In comparison to TCP and User Datagram Protocol (UDP), it offers native support for multihoming. However, because it is not as popular, the Linux kernel does not offer a BPF helper function to recompute the checksum for packets that are redirected. Therefore, Loxilb has to recompute the SCTP checksum in BPF itself. Packet redirection thus consumes much more CPU time and the throughput is reduced by over 60× in comparison to TCP. This also causes increased relative overheads when we activate the Spectre-STL defenses (2.7 %) and VeriFence (14 %). To improve SCTP's performance, it would be the most promising to either disable checksum calculation [95], or create a kernel helper function. Compared to the potential speedup by avoiding checksum recalculation in BPF altogether, the speedup achieved by disabling VeriFence is insignificant for SCTP.

In summary, VeriFence does not bottleneck the TCP and SCTP performance achievable over the load balancer. This is the case especially because the overhead would be even lower if the containers were not colocated on the same machine.

For our final benchmark, we run two `nginx` servers serving a 1 KiB payload and connect to them through a single Loxilb load balancer instance. We use the `wrk2` load generator [104] with two threads to target a specific HTTP request rate and analyze the resulting tail latency (99nth percentile) and average rate. We use `wrk2` because it does not suffer from coordinated omissions. With two servers, the maximum achievable rate is approx. 31 kRPS. Based on this, we scale the target load from 80 % to 105 % in increments of 0.1 % and then group the data into bins of 2.5 %. We run each test for 1 min and repeat the measurement 10 times, therefore each bin contains 250 data points. We omit outliers from this plot.

The right plots of Figure 8 show the achieved average rate and tail latency for different target rates. As expected, we observe increased variation for all configurations when running the system close to 31 kRPS as small variations are easily amplified due to system-overload. Importantly, VeriFence does not cause increased tail latencies. This is because VeriFence merely increases the time needed to process each request by a small amount, but does not cause any unpredictable spikes in the processing time.

8.2.4 Summary. **VeriFence does not affect most of the applications we analyze in a measurable way.** Event tracers appear unaffected, which is particularly important as developers often write small tracing programs ad-hoc [127] and therefore are unlikely to exhaustively review their programs for gadgets. For more complex BPF programs as used by the Parca Continuous Profiler and the Loxilb Network Load Balancer, we find that both the Spectre-STL defenses and VeriFence can affect the CPU time spent executing BPF code, however, **this usually does not impact application performance, showing that it should only be of secondary concern.** Instead, it is much more important to be able to use BPF in the first place. In any case, users can always disable VeriFence per–BPF-program at runtime to trade security for performance.

9 RELATED WORK

In this section, we discuss alternatives to BPF and then relevant alternative techniques to defend against transient execution attacks.

9.1 High-Performance IO

BPF implements safe kernel extensions [45, 61, 136] for Linux. While it has numerous applications outside of high-performance IO [58, 70, 72, 80, 128], asynchronous IO and kernel-bypass can replace it in some cases.

9.1.1 *Asynchronous IO.*

To improve performance over traditional system calls, asynchronous IO [12] is implemented by `aio` [21] and more recently by `io_uring` [15] in Linux [42]. Being closely related to system-call batching [49, 132], it amortizes mode switches over multiple operations. However, both do not improve upon the latency of traditional system-calls because data still has to pass through the operating system (OS) network or storage stack to reach the application. In contrast, BPF allows applications to process (or discard) data directly on the CPU where it is first retrieved [135].

9.1.2 *Kernel-Bypass.*

The most prominent implementations of kernel–bypass are DPDK [88, 103] and SPDK [11, 130]. By giving user applications direct access to the hardware, kernel-bypass can reduce both IO latency and throughput. However, in contrast to BPF, kernel-bypass does not integrate cooperatively with the existing networking stack and requires dedicating full CPU cores to busy-looping for low-latency packet processing (hurting power-proportionality). Further, when kernel-bypass and regular applications are colocated on a server, kernel-bypass reduces performance for regular applications because packets have to be re-injected into the kernel networking stack to reach them [62]. For this reason, Meta reportedly uses a BPF-based load balancer (similar to Loxilb from our evaluation) instead of kernel-bypass [40].

9.2 Transient-Execution–Attack Defenses

To defend against transient execution attacks, VeriFence relies on compiler-based defenses. Alternative approaches partition resources (OS-based) or attempt to implement side-channel–resistant transient execution that is still low-overhead (hardware-based).

9.2.1 *Compiler-based Defenses.*

To the best of our knowledge, Linux's BPF verifier is the only widely deployed sandbox that is fully hardened against the known Spectre vulnerabilities without relying on process isolation. While incomplete Spectre defenses have been implemented for some sandboxes (e.g., index masking and pointer poisoning for JavaScript [101]), all production Java [91], Wasm [102, 122], and JavaScript [31, 117] runtimes still rely on process isolation for full mitigation, as recommended by Intel [63]. However, the kernel cannot apply process isolation to BPF without reducing its performance significantly [49]. The Windows BPF verifier (Prevail [71]) does not implement Spectre defenses as of April 2024 [51].

There exist theoretical works that can potentially detect Spectre gadgets more precisely than the BPF verifier (even with our extension) [36, 100]. However, these cannot be readily applied to BPF because they make simplifying assumptions in their implementation [56] or are too complex to implement [119] (and would, therefore, further increase the risk of security-critical bugs). The common C/C++ toolchains do not offer reliable, architecture-agnostic Spectre defenses that are transparent to the users [1, 18–20, 63, 69, 75], therefore the BPF verifier must apply defenses to the programs. Most related works on BPF focus on architectural security but do not consider transient execution attacks [82, 84, 93, 94]. In future work, the number of barriers VeriFence inserts could be further reduced by using ALU instructions to detect misprediction and erase sensitive data (e.g., using Speculative Load Hardening, SLH [33, 34, 100, 110, 134] and similar techniques [24, 96, 97]).

While browsers to date rely on process isolation to prevent Wasm programs from exploiting Spectre [96, 102], there is some work on compiler-based defenses in specific runtimes. However, we find that they are either incomplete (i.e., Wasmtime [122], V8 [31, 87]) or specific to Wasm/user space (i.e., Swivel [90]) and do not apply to BPF in the kernel.

9.2.2 *OS-based Defenses.*

To defend against transient execution attacks, most works rely on coarse-grained partitioning of resources [59, 60] with support from the OS (e.g., address-space isolation [17] and core scheduling [3]). However, address-space isolation cannot be applied to BPF without limiting its performance even further [49]. Applying Memory Protection Keys (MPKs) to BPF against Spectre appears to be a promising direction for future research [2, 63]. Existing work on the topic does not take transient execution attacks into account [82] but assumes the BPF verifier (e.g., our work) implements defenses.

9.2.3 *Hardware-based Defenses.*

As of April 2024, there exists no practical high-performance processor implementation that is not vulnerable to Spectre [8–10]. Instead, vendors continue to recommend compiler-based defenses against all cross- and in-domain transient execution attacks [2, 65]. To date, complete protection from microarchitectural timing side-channels is not possible without significant changes to the instruction set architecture (ISA) and processor design [29, 57, 126, 131].

10 CONCLUSION

This work has presented VeriFence for Linux BPF, a practical and easy-to-apply enhancement to the only software-based sandbox resilient against Spectre. VeriFence is *sound*, *precise*, and *lightweight*. First, it *soundly* combines speculation barriers and static analysis in a way that does not increase the kernel's attack surface. Second, it *precisely* prevents unsafe transient behavior thereby reducing the number of rejected programs from 31 % to 4 % (with only test programs from the Linux selftests remaining). Finally, as our real-world performance evaluation demonstrates, it is *lightweight* because it does not impact BPF invocation latency, which is of particular importance to BPF as it complements user space by offering minimum-overhead transitions.

ACKNOWLEDGMENTS

We thank Daniel Borkmann, Alexei Starovoitov, Benedict Schlüter, Marco Guarnieri, Georg Julius Liefke, and Anil Kurmus for taking the time to review our kernel patches and giving their thoughtful feedback on the project. We are grateful to the anonymous reviewers and our shepherd for their insightful comments. This work is supported by the German Research Foundation (DFG) – project numbers 539710462 (DOSS), 502615015 (ResPECT), 502228341 (Memento), and 465958100 (NEON).

REFERENCES

[1] 2021. Visual Studio 2022: C++ Developer Guidance for Speculative Execution Side Channels. https://learn.microsoft.com/en-us/cpp/security/developer-guidance-speculative-execution?view=msvc-170

[2] 2022. *Refined Speculative Execution Terminology*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html

[3] 2023. *Core Scheduling – The Linux Kernel documentation (v6.5)*. https://www.kernel.org/doc/html/v6.5/admin-guide/hw-vuln/core-scheduling.html

[4] 2023. *Debian Manpages – bpf(2)*. https://manpages.debian.org/bookworm/manpages-dev/bpf.2.en.html

[5] 2023. *eBPF verifier – The Linux Kernel documentation (v6.5)*. https://www.kernel.org/doc/html/v6.5/bpf/verifier.html We use *BPF* to refer to the current extended Berkeley Packet Filter facility.

[6] 2023. *HID-BPF – The Linux Kernel documentation (v6.5)*. https://www.kernel.org/doc/html/v6.5/hid/hid-bpf.html

[7] 2023. *kernel/bpf/syscall.c – Line 2588 – Linux Kernel Stable Tree – v6.5.11*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/bpf/syscall.c?id=79944183#n2588

[8] 2023. *Software Techniques for Managing Speculation on AMD Processors - Revision 5.09.23*. https://web.archive.org/web/20240313145139/https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/software-techniques-for-managing-speculation.pdf

[9] 2023. *Speculative Processor Vulnerability (Updated December 20, 2023) - Arm Security Center*. https://web.archive.org/web/20240324073901/https://developer.arm.com/ArmSecurityCenter/SpeculativeProcessorVulnerability

[10] 2024. *Affected Processors: Guidance for Security Issues on Intel® Processors*. https://web.archive.org/web/20240411104155/https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html

[11] 2024. *SPDK: Storage Performance Development Kit*. https://spdk.io/

[12] Alexander van der Grinten. 2024. *The Managarm Project*. https://managarm.org/

[13] ARM. 2018. Firmware interfaces for mitigating cache speculation vulnerabilities – System Software on Arm Specification (Version 1.3). https://developer.arm.com/cache-speculation-vulnerability-firmware-specification

[14] ARM. 2021. *ARM Documentation – DIT, Data Independent Timing – Arm Armv8-A Architecture Registers*. https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/DIT--Data-Independent-Timing

[15] Jens Axboe. 2019. *[PATCHSET v5] io_uring IO interface (Mail)*. https://lore.kernel.org/linux-block/20190116175003.17880-1-axboe@kernel.dk/

[16] Pavel Begunkov. 2021. *io_uring: BPF controlled I/O*. https://lpc.events/event/11/contributions/901/

[17] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M Frans Kaashoek, and Nickolai Zeldovich. 2020. Efficiently mitigating transient execution attacks using the unmapped speculation contract. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI'20)* (2020). USENIX, 1139–1154. https://www.usenix.org/conference/osdi20/presentation/behrens

[18] Kristof Beyls. 2018. *D41760: Introduce __builtin_load_no_speculate*. https://reviews.llvm.org/D41760

[19] Kristof Beyls. 2018. *D49070: Introduce llvm.speculation_safe_value intrinsic*. https://reviews.llvm.org/D49070

[20] Kristof Beyls. 2018. *D49073 from the LLVM Phabricator archive: Introducing __builtin_speculation_safe_value*. https://reviews.llvm.org/D49073

[21] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. 2003. *Asynchronous I/O Support in Linux 2.5*. https://www.kernel.org/doc/ols/2003/ols2003-pages-351-366.pdf

[22] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: : Exploiting Speculative Execution through Port Contention. In *Proceedings of the 26th SIGSAC Conference on Computer and Communications Security (CCS'19)*. ACM, 785–800. https://doi.org/10.1145/3319535.3363194

[23] Daniel Borkmann. 2021. *BPF and Spectre: Mitigating transient execution attacks*. https://www.youtube.com/watch?v=6N30Yp5f9c4 eBPF Summit 2021.

[24] Daniel Borkmann, Jann Horn, and Alexei Starovoitov. 2019. *Linux Kernel Source Tree – bpf: prevent out of bounds speculation on pointer arithmetic (commit #979d63d5)*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=979d63d5

[25] Daniel Borkmann, Piotr Krysiuk, John Fastabend, and Alexei Starovoitov. 2021. *Linux Kernel Source Tree – bpf: Fix leakage of uninitialized bpf stack under speculation (commit #801c60)*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=801c6058

[26] Daniel Borkmann, Piotr Krysiuk, Benedict Schlueter, John Fastabend, and Alexei Starovoitov. 2021. *Linux Kernel Source Tree – bpf: Tighten speculative pointer arithmetic mask (commit #7fedb6)*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7fedb63a

[27] Daniel Borkmann, Piotr Krysiuk, Benedict Schlueter, and Alexei Starovoitov. 2021. *Linux Kernel Source Tree – bpf: Fix leakage due to insufficient speculative store bypass mitigation (commit #2039f2)*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2039f26f

[28] Daniel Borkmann, Adam Morrison, Ofek Kirzner, Benedict Schlüter, Piotr Krysiuk, John Fastabend, and Alexei Starovoitov. 2021. *Linux Kernel Source Tree – bpf: Fix leakage under speculation on mispredicted branches (commit #918367)*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9183671a

[29] Scott Buckley, Robert Sison, Nils Wistoff, Curtis Millar, Toby Murray, Gerwin Klein, and Gernot Heiser. 2023. Proving the Absence of Microarchitectural Timing Channels. https://doi.org/10.48550/arXiv.2310.17046

[30] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*. USENIX, 991. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[31] Mathias Bynens. 2018. *V8 Docs – Untrusted code mitigations*. https://v8.dev/docs/untrusted-code-mitigations#sandbox-untrusted-execution-in-a-separate-process

[32] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. USENIX, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[33] Chandler Carruth. 2018. *[llvm-dev] RFC: Speculative Load Hardening (a Spectre variant #1 mitigation)*. https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html

[34] Chandler Carruth. 2024. Speculative Load Hardening — LLVM 16.0.0git documentation. https://llvm.org/docs/SpeculativeLoadHardening.html

[35] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. ACM, 913–926. https://doi.org/10.1145/3385412.3385970

[36] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *Proceedings of the 43rd Symposium on Security and Privacy (SP'22)*. IEEE, 666–680. https://doi.org/10.1109/SP46214.2022.9833707

[37] Sunjay Cauligi, Marco Guarnieri, Daniel Moghimi, Deian Stefan, and Marco Vassena. 2022. A Turning Point for Verified Spectre Sandboxing. https://doi.org/10.48550/arXiv.2208.01548

[38] Shaun Crampton. 2020. *Introducing the Calico eBPF dataplane*. https://www.tigera.io/blog/introducing-the-calico-ebpf-dataplane/

[39] Milo Craun, Adam Oswald, and Dan Williams. 2023. Enabling eBPF on Embedded Systems Through Decoupled Verification. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions (eBPF'23)*. ACM, 63–69. https://doi.org/10.1145/3609021.3609299

[40] Nikita Shirokov Dasineni, Ranjeeth. 2018. *Engineering at Meta – Open-sourcing Katran, a scalable network load balancer*. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

[41] Jules Dejaeghere, Bolaji Gbadamosi, Tobias Pulls, and Florentin Rochet. 2023. Comparing Security in eBPF and WebAssembly. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions (eBPF'23)*. ACM, 35–41. https://doi.org/10.1145/3609021.3609306

[42] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR'22)*. ACM, 120–127. https://doi.org/10.1145/3534056.3534945

[43] Jake Edge. 2022. *Progress for unprivileged containers*. https://lwn.net/Articles/909627/

[44] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. 2022. Automatic Detection of Speculative Execution Combinations. https://doi.org/10.48550/arXiv.2209.01179

[45] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. 2006. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of the 1st European Conference on Computer Systems (EuroSys'06)*. ACM, 177–190. https://doi.org/10.1145/1217935.1217953

[46] Luis Gerhorst. 2023. *[PATCH 2/3] Revert "bpf: Fix issue in verifying allow_ptr_leaks"*. https://lore.kernel.org/bpf/20230913122827.91591-1-gerhorst@amazon.de/

[47] Luis Gerhorst. 2023. *Re: [PATCH 2/3] Revert "bpf: Fix issue in verifying allow_ptr_leaks"*. https://lore.kernel.org/bpf/20230928110927.115238-1-gerhorst@amazon.de/

[48] Luis Gerhorst and Daniel Borkmann. 2023. *Linux Kernel Source Tree – bpf: Remove misleading spec_v1 check on var-offset stack read (commit #082cdc)*. https://git.

kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=082cdc69

[49] Luis Gerhorst, Benedict Herzog, Stefan Reif, Wolfgang Schröder-Preikschat, and Timo Hönig. 2021. AnyCall: Fast and Flexible System-Call Aggregation. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS'21)*. ACM, 1–8. https://doi.org/10.1145/3477113.3487267

[50] Luis Gerhorst, Henriette Hofmeier, and Daniel Borkmann. 2023. *Linux Kernel Source Tree – bpf: Fix pointer-leak due to insufficient speculative store bypass mitigation (commit #e4f4db47)*. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/commit/?id=e4f4db47

[51] Elazar Gershuni. 2022. *Use verifier to identify dependent reads that require speculative load hardening (GitHub, Issue #229, vbpf/ebpf-verifier)*. https://github.com/vbpf/ebpf-verifier/issues/229

[52] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, 1069–1084. https://doi.org/10.1145/3314221.3314590

[53] Yossi Gottlieb, YaacovHazan (GitHub User), Filipe Oliveira, and Oran Agra. 2022. *memtier_benchmark – v1.4.0 – NoSQL Redis and Memcache traffic generation and benchmarking tool*. https://github.com/RedisLabs/memtier_benchmark/tree/29f51a82

[54] Thomas Graf, André Martins, Joe Stringer, and Paul Chaignon. 2021. *CNI Performance Benchmark — Cilium documentation*. https://docs.cilium.io/en/v1.12/operations/performance/benchmark/

[55] Brendan Gregg. 2016. The flame graph. *Commun. ACM* 59, 6 (May 2016), 48–57. https://doi.org/10.1145/2909476

[56] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *Proceedings of the 41st Symposium on Security and Privacy (SP'20)*. IEEE, 1–19. https://doi.org/10.1109/SP40000.2020.00011

[57] Gernot Heiser, Toby Murray, and Gerwin Klein. 2020. Towards Provable Timing-Channel Prevention. 54, 1 (2020), 1–7. https://doi.org/10.1145/3421473.3421475

[58] Tejun Heo. 2023. *[PATCHSET v3] sched: Implement BPF extensible scheduler class*. https://lore.kernel.org/all/20230317213333.2174969-1-tj@kernel.org/

[59] Mathé Hertogh, Manuel Wiesinger, Sebastian Österlund, Marius Muench, Nadav Amit, Herbert Bos, and Cristiano Giuffrida. 2023. Quarantine: Mitigating Transient Execution Attacks with Physical Domain Isolation. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'23)*. ACM, 207–221. https://doi.org/10.1145/3607199.3607248

[60] Henriette Hofmeier. 2022. Dynamic Reconfiguration of Hardware-Vulnerability Mitigations in the Linux Kernel. https://sys.cs.fau.de/publications/2022/hofmeier_22_thesis.pdf

[61] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. 41, 2 (2007), 37–49. https://doi.org/10.1145/1243418.1243424

[62] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, 54–66. https://doi.org/10.1145/3281411.3281443

[63] Intel. 2018. *Managed Runtime Speculative Execution Side Channel Mitigations*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/runtime-speculative-side-channel-mitigations.html

[64] Intel. 2018. *Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html

[65] Intel. 2020. An optimized mitigation approach for Load Value Injection. https://www.intel.com/content/www/us/en/develop/articles/software-security-guidance/best-practices/optimized-mitigation-approach-load-value-injection.html

[66] Intel. 2023. *Data Operand Independent Timing Instructions*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html

[67] Intel. 2023. *Data Operand Independent Timing ISA Guidance*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html

[68] Javier Honduvilla Coto. 2023. *parca-dev/parca-agent (GitHub) – v0.27.0*. https://github.com/parca-dev/parca-agent

[69] Devin Jeanpierre and Chandler Carruth. 2020. Mitigating Spectre v1 Attacks in C++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0928r1.pdf

[70] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. 2023. Programmable System Call Security with eBPF. https://doi.org/10.48550/arXiv.2302.10366

[71] Alan Jowett. 2021. *(PREVAIL) Spectre Mitigations Issue (GitHub, Issue #229, vbpf/ebpf-verifier)*. https://github.com/vbpf/ebpf-verifier/issues/229

[72] Daniel Kim and Robert Prast. 2022. *Triaging Real-time Security Threats with eBPF-powered Observability*. https://www.usenix.org/conference/srecon22americas/presentation/kim

[73] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In *Proceedings of the 30th SIGSAC Conference on Computer and Communications Security (CCS'23)*. ACM, 2038–2052. https://doi.org/10.1145/3576915.3616611

[74] Ofek Kirzner and Adam Morrison. 2021. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*. USENIX, 2399–2416.

[75] Paul Kocher. 2018. *Spectre Mitigations in Microsoft's C/C++ Compiler*. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[76] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th Symposium on Security and Privacy (SP'19)*. IEEE, 1–19. https://doi.org/10.1109/SP.2019.00002

[77] Greg Kroah-Hartman, Linus Torvalds, David S. Miller, and Arnd Bergmann. 2023. *Linux Kernel Stable Tree – v6.5.11*. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?id=79944183 BPF Samples from samples/bpf and Kernel Selftests from tools/testing/selftests/bpf..

[78] Piotr Krysiuk, Benedict Schlüter, and Daniel Borkmann. 2022. *BPF and Spectre: Mitigating transient execution attacks*. https://popl22.sigplan.org/details/prisc-2022-papers/11/BPF-and-Spectre-Mitigating-transient-execution-attacks Keynote at PriSC'22.

[79] Chris Lamb, dormando (GitHub User), Dustin Sallings, and Brad Fitzpatrick. 2020. *memcached – v1.6.9*. https://packages.debian.org/bullseye/web/memcached

[80] Zhilu Lian, Yangzi Li, Zhixiang Chen, Shiwen Shan, Baoxin Han, and Yuxin Su. 2022. eBPF-based Working Set Size Estimation in Memory Management. , 188–195 pages. https://doi.org/10.1109/ICSS55994.2022.00036

[81] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*. USENIX, 973–990. https://doi.org/10.1145/3357033

[82] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. 2023. MOAT: Towards Safe BPF Kernel Extension. https://doi.org/10.48550/arXiv.2301.13421

[83] Bruce A. Mah, Jef Poskanzer, Jon Dugan, and Brian Tierney. 2020. *iperf3 – v3.9 (cJSON 1.7.13)*. https://packages.debian.org/bullseye/iperf3

[84] Sai Veerya Mahadevan, Yuuki Takano, and Atsuko Miyaji. 2021. PRSafe: Primitive Recursive Function based Domain Specific Language using LLVM. In *Proceedings of the 20th International Conference on Electronics, Information, and Communication (ICEIC'21)*. IEEE, 1–4. https://doi.org/10.1109/ICEIC51217.2021.9369763

[85] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. 2019. Speculator: a tool to analyze speculative execution attacks and mitigations. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*. ACM, 747–761. https://doi.org/10.1145/3359789.3359837

[86] Andrei Matei and Alexei Starovoitov. 2021. *Linux Kernel Source Tree– bpf: Allow variable-offset stack access (commit #01f810)*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit?id=01f810ac

[87] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. https://doi.org/10.48550/arXiv.1902.05178

[88] Thomas Monjalon, Bruce Richardson, Qi Z. Zhang, and Stephen Hemminger. 2024. *DPDK: Data Plane Development Kit – v24.03*. http://git.dpdk.org/dpdk/tree/?h=v24.03&id=a9778aad

[89] Andrii Nakryiko, waruqi (GitHub User), Daniel Müller, and Hengqi Chen. 2022. *libbpf/libbpf-bootstrap (GitHub, commit #a7c0f7)*. https://github.com/libbpf/libbpf-bootstrap/tree/a7c0f7e4

[90] Narayan, Disselkoen, Moghimi, and others. 2021. Swivel: Hardening WebAssembly against Spectre. In *Proceeding of the 30th USENIX Security Symposium (USENIX Security '21)*. USENIX, 1433–1450. https://www.usenix.org/conference/usenixsecurity21/presentation/narayan

[91] Amir Naseredini, Stefan Gast, Martin Schwarzl, Pedro Miguel Sousa Bernardo, Amel Smajic, Claudio Canella, Martin Berger, and Daniel Gruss. 2021. Systematic Analysis of Programming Languages and Their Execution Environments for Spectre Attacks. https://doi.org/10.48550/arXiv.2111.12528

[92] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 41–61. https://www.usenix.org/conference/osdi20/presentation/nelson

[93] Luke Nelson and Xi Wang. 2022. *Exoverifier (GitHub, commit #a0166f6)*. https://github.com/uw-unsat/exoverifier/tree/a0166f6

[94] Luke Nelson, Xi Wang, and Emina Torlak. 2021. *A proof-carrying approach to building correct and flexible in-kernel verifiers.* https://homes.cs.washington.edu/~lukenels/slides/2021-09-23-lpc21.pdf Linux Plumbers Conference (LPC'21).

[95] nik-netlox (GitHub User). 2023. *SCTP: >10x BPF program runtime compared to TCP (GitHub, Issue #447, loxilb).* https://github.com/loxilb-io/loxilb/issues/447#issuecomment-1832999598

[96] Ejebagom John Ojogbo, Mithuna Thottethodi, and T N Vijaykumar. 2020. Secure automatic bounds checking: prevention is simpler than cure. In *Proceedings of the 18th International Symposium on Code Generation and Optimization (CGO'20).* ACM, 42–55. https://doi.org/10.1145/3368826.3377921

[97] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, Christof Fetzer, and T U Dresden. 2018. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. https://doi.org/10.48550/arXiv.1805.08506v3

[98] Tomasz Osiński. 2020. *p4c-ubpf: a New Back-end for the P4 Compiler.* https://opennetworking.org/news-and-events/blog/p4c-ubpf-a-new-back-end-for-the-p4-compiler/

[99] PacketCrunch (GitHub User), TrekkieCoder (GitHub User), nik-netlox (GitHub User), and UltraInstinct14 (GitHub User). 2024. *loxilb: eBPF based cloud-native load-balancer (GitHub) – v0.9.0.* https://github.com/loxilb-io/loxilb

[100] Marco Patrignani and Marco Guarnieri. 2021. Exorcising spectres with secure compilers. In *Proceedings of the 28th SIGSAC Conference on Computer and Communications Security (CCS'21).* ACM, 445–461. https://doi.org/10.1145/3460120.3484534

[101] Filip Pizlo. 2018. *What Spectre and Meltdown Mean For WebKit.* https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/

[102] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19).* USENIX, 1661–1678. https://www.usenix.org/conference/usenixsecurity19/presentation/reis

[103] Rami Rosen. 2017. *Network acceleration with DPDK [LWN.net].* https://lwn.net/Articles/725254/

[104] Ragnar Rova, Will Glozer, Gil Tene, and Michael Barker. 2023. *wrk2 (GitHub, commit #920ce1e).* https://github.com/rrva/wrk2/tree/920ce1e

[105] Benedict Schlüter. 2021. Security Analysis of eBPF. Bachelor's Thesis. Ruhr-Universität Bochum (RUB).

[106] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross–Privilege–Boundary Data Sampling. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19).* ACM, 753–768. https://doi.org/10.1145/3319535.3354252

[107] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. NetSpectre: Read arbitrary memory over network. https://doi.org/10.48550/arXiv.1807.10535

[108] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy* (Phoenix, AZ, USA) *(HASP '19).* Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3337167.3337175

[109] Yafang Shao. 2023. *Re: [PATCH 2/3] Revert "bpf: Fix issue in verifying allow_ptr_leaks".* https://patchwork.kernel.org/project/linux-kselftest/patch/20230913122827.91591-1-gerhorst@amazon.de/#25515964

[110] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *Proceedings of the 44th Symposium on Security and Privacy (SP'23).* IEEE, 1753–1770. https://doi.org/10.1109/SP46215.2023.10179355

[111] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Gregoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *Proceedings of the 43rd Symposium on Security and Privacy (SP'23).* IEEE, 1094–1111. https://doi.org/10.1109/SP46215.2023.10179418

[112] Yonghong Song, Brendan Gregg, Sasha Goldshtein, and Teng Qin. 2023. *BCC – v0.27.0.* https://github.com/iovisor/bcc/tree/v0.27.0 We use the libbpf-based variants from the libbpf-tools folder.

[113] Alexei Starovoitov. 2023. *Re: [PATCH 2/3] Revert "bpf: Fix issue in verifying allow_ptr_leaks".* https://patchwork.kernel.org/project/linux-kselftest/patch/20230913122827.91591-1-gerhorst@amazon.de/#25516346

[114] Alexei Starovoitov, John Fastabend, and Daniel Borkmann. 2018. *Linux Kernel Source Tree – bpf: prevent out-of-bounds speculation (commit #b2157399cc98).* https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b2157399cc98

[115] Alexei Starovoitov, Joanne Koong, Andrii Nakryiko, and Jakub Kicinski. 2023. *kernel/bpf/verifier.c – Line 644 – Linux Kernel Stable Tree – v6.5.11.* https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/bpf/verifier.c?id=79944183#n644

[116] Joe Stringer, Jussi Mäki, André Martins, and Thomas Graf. 2022. *cilium/cilium (GitHub) – v1.12.5.* https://github.com/cilium/cilium/tree/v1.12.5

[117] Ben L. Titzer and Jaroslav Sevcik. 2019. *A year with Spectre: a V8 perspective.* https://v8.dev/blog/spectre

[118] Linux Torvalds, Delyan Kratunov, Jiri Olsa, and Milan Landaverde. 2022. *bpftool – v5.18 (Linux kernel release).* https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/tools/bpf/bpftool?h=v5.18 We use an older version because the latest version does not support legacy map definitions that are still used by many projects. This does not impact the in-kernel verification and JIT compilation (and thereby our results).

[119] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. 5 (2021), 1–30. Issue POPL. https://doi.org/10.1145/3434330

[120] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. 53, 1 (2020), 16:1–16:36. https://doi.org/10.1145/3371038

[121] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, precise, and fast abstract interpretation with tristate numbers. In *Proceedings of the 20th International Symposium on Code Generation and Optimization (CGO'22).* IEEE, 254–265. https://doi.org/10.1109/CGO53902.2022.9741267

[122] Wasmtime. 2024. *Wasmtime Docs – Security.* https://docs.wasmtime.dev/security.html#spectre

[123] Erik Wenzel, Grant Grundler, Gavin Brebner, and Marcelo Ricardo Leitner. 2020. *netperf – v2.7.0.* https://packages.debian.org/bullseye/netperf

[124] Hadley Wickham, Winston Chang, Lionel Henry, and Thomas Lin Pedersen. 2024. *ggplot2 3.5.0: A box and whiskers plot (in the style of Tukey) — geom_boxplot.* https://ggplot2.tidyverse.org/reference/geom_boxplot.html

[125] Rafael J. Wysocki. 2023. *CPU Performance Scaling – The Linux Kernel documentation (v6.5).* https://www.kernel.org/doc/html/v6.5/admin-guide/pm/cpufreq.html#generic-scaling-governors

[126] Wenjie Xiong and Jakub Szefer. 2021. Survey of Transient Execution Attacks and Their Mitigations. *ACM Comput. Surv.* 54, 3, Article 54 (5 2021), 36 pages. https://doi.org/10.1145/3442479

[127] Daniel Xu, fbs (GitHub User), Alastair Robertson, and Viktor Malík. 2022. *bpftrace – v0.20.3.* https://github.com/bpftrace/bpftrace/tree/v0.20.3

[128] Junqing Yang, Lijun Chen, and Jiaqing Bai. 2022. Redis automatic performance tuning based on eBPF. In *Proceedings of the 14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA'22).* IEEE, 671–676. https://doi.org/10.1109/ICMTMA54903.2022.00139

[129] Rui Yang and Marios Kogias. 2023. HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions* (New York, NY, USA) *(eBPF '23).* Association for Computing Machinery, New York, NY, USA, 77–83. https://doi.org/10.1145/3609021.3609307

[130] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *Proceedings of the 9th International Conference on Cloud Computing Technology and Science (CloudCom'17).* IEEE, 154–161. https://doi.org/10.1109/CloudCom.2017.14

[131] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52).* Association for Computing Machinery, New York, NY, USA, 954–968. https://doi.org/10.1145/3352460.3358274

[132] E. Zadok, S. Callanan, Abhishek Rai, Gopalan Sivathanu, and A. Traeger. 2005. Efficient and safe execution of user-level code in the kernel. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS'05).* IEEE. https://doi.org/10.1109/IPDPS.2005.189

[133] Ioannis Zarkadas, Tal Zussman, Jeremy Carin, Sheng Jiang, Yuhong Zhong, Jonas Pfefferle, Hubertus Franke, Junfeng Yang, Kostis Kaffes, Ryan Stutsman, and Asaf Cidon. 2023. BPF-oF: Storage Function Pushdown Over the Network. https://doi.org/10.48550/arXiv.2312.06808

[134] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2022. Breaking and Fixing Speculative Load Hardening. https://www.cryptojedi.org/papers/uslh-20220605.pdf

[135] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22).* USENIX Association, Carlsbad, CA, 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong

[136] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. 2023. Userspace Bypass: Accelerating Syscall-intensive Applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23).* USENIX Association, Boston, MA, 33–49. https://www.usenix.org/conference/osdi23/presentation/zhou-zhe