Firefly: High-Performance Transient Execution Attack Defenses for In-Kernel Sandboxes

Luis Gerhorst[1,2][0000−0002−3401−430X], Henriette Herzog[2][0000−0002−0828−6862], Benedict Herzog[2][0000−0001−8725−3454], Peter Wägemann[1][0000−0002−3730−533X], Rüdiger Kapitza[1][0000−0002−8116−7763], and Timo Hönig[2][0000−0002−1818−0869]

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
luis.gerhorst@fau.de, waegemann@cs.fau.de, ruediger.kapitza@fau.de
[2] Ruhr-Universität Bochum (RUB), Germany
henriette.herzog@rub.de, benedict.herzog@rub.de, timo.hoenig@rub.de

Abstract. Safe operating system kernel extensions, as implemented by Linux eBPF, enable improved application performance, kernel observability, and system security. To extend these benefits beyond privileged applications, the kernel must ensure that eBPF programs cannot escape their in-kernel sandbox through transient execution attacks, such as Spectre. However, all existing defenses for eBPF (i.e., upstream eBPF, VeriFence, BeeBox, and HIVE) lack at least one of three properties an in-kernel sandbox requires: (1) low overhead, (2) portability, and (3) address-space sharing. This raises the systems design question of whether in-kernel sandboxes must give up one of these properties to defend against transient execution attacks.

To answer this question, we present Firefly, a verifier-based defense for eBPF that simultaneously achieves these three properties through co-design of verifier logic, helper interfaces, and code generation. At its core is a speculation-aware verifier that unifies Spectre-PHT (v1) and Spectre-STL (v4) handling while using a refined constant-time leakage model that also protects physical-layout information. Unlike BeeBox and HIVE, Firefly does not require ARM64-specific hardware features, address-space separation, or driver-specific optimizations. On x64, Firefly matches the most performant, but use-case–specific, BeeBox variant, whereas generic state-of-the-art alternatives add 71 % to 151 % for Meta's Katran load balancer. Firefly, in contrast, only induces 6 % to 26 % eBPF execution-time overhead.

Keywords: Linux, Spectre, eBPF, Systems Design

1 Introduction

Linux's extended Berkeley Packet Filter (eBPF) [63, 73], the most prominent implementation of safe kernel extensions, enables very low-overhead interaction between user and kernel space. eBPF's applications to date include container networking (as used by Docker and Kubernetes), high-performance input/output (IO) [54,88,114], energy [95] and carbon monitoring [10,93], 5G net-

works [101], memory optimization [71], threat detection [45,64], system call filtering [59], automatic performance tuning [124], (energy-aware) scheduling [50,94], and databases [126, 128]. Because of its success, Microsoft has ported eBPF to Windows [62], and the FreeBSD community is working to support it as well [125]. Furthermore, an IETF working group aims to standardize the technology [113].

eBPF supports these applications by enabling low-overhead interaction between user and kernel code. It avoids domain transitions by just-in-time compiling user-provided bytecode and running that code in kernel mode at near-native speed. To isolate eBPF programs from the kernel, the verifier uses abstract interpretation to check memory and type safety before execution. This check imposes some restrictions, but it allows for good expressiveness and near-native performance. To use eBPF, most applications today require root privileges (*privileged eBPF*), but Linux also allows unprivileged user processes to load safety-checked bytecode into the kernel (*unprivileged eBPF*). In summary, eBPF is currently the leading technology for safe kernel extensions [58, 61, 68].

Driven by eBPF's numerous applications in today's critical infrastructure, this work focuses on improving eBPF's isolation from the kernel by enabling low-overhead and architecture-agnostic defenses against transient execution attacks. The consequences are two-fold. Either one can use this to improve the system's security by not requiring elevated privileges for processes that already use eBPF [29,76], or to improve the system's performance by using eBPF in new applications. Upstream Linux already uses unprivileged eBPF for network traffic filters [41,103], and potential future use cases include asynchronous I/O [19,32,69], stateful system-call filters [59], (human-interface) device drivers [109], file monitoring [34], system-call aggregation [43], and containers [37, 55].

To support these unprivileged applications on modern out-of-order CPUs, eBPF must implement low-overhead, software-based defenses against transient execution attacks. Whereas hardware can efficiently prevent Meltdown and similar domain-bypass transient execution attacks (e.g., where the hardware crosses address-space boundaries [3]), the cross- and in-domain transient execution attacks that researchers have uncovered since 2018 (e.g., Spectre [26, 66]) still require software defenses [33, 56, 79] on all high-performance CPUs [6–8]. This is required to prevent eBPF-capable applications from accessing arbitrary kernel and user data. To protect the system, eBPF must apply defenses to the machine code it generates from the application's bytecode.

To date, all existing defenses for eBPF (VeriFence [44], BeeBox [60], and HIVE [127]) either suffer from performance, portability, or sharing issues. Furthermore, combining the different approaches to resolve this is not viable. First, VeriFence, like upstream eBPF, suffers from high performance overhead due to speculation barriers inserted by two separate mechanisms: type analysis for Spectre-STL [1, 26] and speculative path analysis for Spectre-PHT [66]. Second, HIVE requires ARM64 Pointer Authentication Codes (PAC) and Load/Store Unprivileged (LSU) instructions, thereby lacking portability. Finally, BeeBox requires a separate eBPF address space, thereby inducing compatibility issues: BeeBox must statically assign allocations to eBPF programs for maximum per-

formance, and developers can no longer directly reuse kernel abstractions in eBPF helper functions. These limitations are well recognized [9, 53, 58, 106], but to date have lacked a solution.

Contributions. The related works' limitations leave an open systems question for in-kernel sandboxes: must in-kernel sandboxes give up low overhead, portability, or address-space sharing to defend against transient execution attacks? Firefly[3] shows that the answer is no. Its key observation is that prior verifier-based defenses are limited not by verification or speculation barriers themselves, but by the criterion used to classify speculative states as well as existing eBPF interface and compiler design choices. We show that these can be addressed in a portable and scalable manner, making a shared-address-space, verifier-based solution viable even for Spectre-STL. Our contributions are threefold:

Problem analysis: We systematize why prior eBPF transient execution attack defenses fail to achieve low overhead, portability, and address-space sharing.

Security criterion and verifier design: We refine speculative constant-time to protect physical memory layout information and present a speculation-aware verifier for eBPF that unifies Spectre-PHT and Spectre-STL.

Practical system support: Based on the identified bottlenecks, we introduce the interface, compiler, and kernel changes needed to make this verifier-based design feasible in practice.

We evaluate Firefly on x64 and ARM64 using real-world eBPF applications, reproduce prior work, and show that Firefly matches or exceeds alternative approaches without their trade-offs.

Key findings. We make the following key findings:

- Spectre-PHT and Spectre-STL can be handled within a single verifier framework rather than by disjoint mechanisms. This unification reduces conceptual and implementation complexity while allowing defenses to be placed based on one consistent leakage criterion.
- A comprehensive verifier-based defense must account not only for classical speculative leakage as captured by speculative constant-time, but also for physical memory layout information leaked through half-Spectre [36]. In particular, benign-looking speculative out-of-bounds accesses can still reveal layout information that hardens Rowhammer attacks.
- Despite this broader scope, speculation-aware verification remains portable and scales to large real-world programs. We show that improved accuracy must not come at the cost of making load-time verification infeasible.
- Therefore, a previously unachieved design point for in-kernel sandboxes is feasible: low-overhead transient execution attack defenses can support portability and address-space sharing. Prior work suggested that system designers had to give up one of these properties, but we show that this trade-off is not required.

[3] Name associates eBPF's bee mascot with spectres

- Achieving this design point requires co-design across verifier logic, helper interfaces, and compiler configuration. Verifier accuracy alone is not sufficient because the used interfaces and source compiler determine how often speculative gadgets arise and how costly their defenses become.

Open science. Parts of our design have been adopted into the 6.17 upstream Linux kernel,[4] and we publish Firefly's full implementation, evaluation, and raw data under an open-source license.[5]

2 Background

2.1 Transient Execution Vulnerabilities

This work focuses on Spectre-PHT [66] and Spectre-STL [1,87] because they are the longest-standing and most performance-critical issues [26,28] to which new processors are still vulnerable [6–8]. Spectre-PHT includes all transient execution vulnerabilities based on conditional branches, as these use the Pattern History Table (PHT) for prediction. Spectre-STL exploits the fact that loads may return stale values due to in-flight stores and misprediction in the CPU's Store To Load (STL) buffer. Because the CPU resolves this ambiguity per load, a store may be bypassed for one load while it is forwarded to another load of the same address [87, Appendix A.6].

We thereby focus on cross-domain and in-domain transient execution attacks [3] but exclude domain-bypass transient execution attacks. The latter include Meltdown [74], Microarchitectural Data Sampling [99], and L1 Terminal Fault [25,51] which can all be efficiently addressed in hardware. Because indirect branches, and thereby Spectre-BTB (v2) gadgets, are far less common in eBPF than direct branches and stores, we rely on existing defenses to prevent Spectre-BTB exploits (e.g., x64 retpoline [20, 67, 110], `IPRED_DIS_S` [57], control-flow integrity [38,42,123] or ARM64 Switchpoline [52]) [121]. Furthermore, Spectre-SLS [11,13] defenses do not affect performance [13,31,40,70].

2.2 Extended Berkeley Packet Filter (eBPF)

Linux eBPF sandboxes user-supplied bytecode and enables its low-overhead invocation in kernel mode. At compile time, compilers generate eBPF bytecode from high-level languages such as C or Rust, and the eBPF toolchain then embeds it into the application binary. At load-time, the application transfers the eBPF bytecode into the kernel using the `bpf()` system call, where the eBPF *verifier* checks its memory and type-safety (and applies transient execution attack defenses). Thereafter, the JIT compiler translates the bytecode to machine code. At runtime, the kernel may invoke eBPF programs in response to various application-specific events (e.g., the arrival of a network packet), and the

[4] `https://lore.kernel.org/r/20250603205800.334980-1-luis.gerhorst@fau.de`
[5] `https://doi.org/10.5281/zenodo.21722512`

programs can communicate with user space via typed shared-memory data structures (eBPF *maps*). To access maps and invoke privileged kernel functions, eBPF programs can call trusted *helpers* that reside in the upstream Linux kernel C source code. These resemble system calls from user space but incur lower latency.

To isolate untrusted extensions, the verifier performs abstract interpretation of all possible program paths to ensure memory and type safety. It starts from the first instruction and spawns new paths whenever it encounters a branch instruction. For abstract values, it distinguishes between scalar types, for which it tracks possible values using multiple abstract domains (e.g., tristate numbers [115]), and pointer types, for which it restricts offsets and dereferencing. The verifier treats pointer values as kernel secrets, while scalar values remain public to the eBPF program. Arguments to helpers must match the argument type, and the verifier assumes helper implementations handle pointer arguments using only constant time (CT) operations (except for dereferencing them). If the verifier detects any potential accesses to kernel-private data, uninitialized registers, out-of-bounds accesses, or type errors, it rejects the eBPF program, thereby preventing the application from using eBPF until application developers rewrite the program to remove the potentially unsafe behavior.

Even without VeriFence [44], BeeBox [60], or HIVE [127], the 6.16 upstream Linux kernel, in principle, fully prevents Spectre-PHT and Spectre-STL on x64 and ARM64. However, while its defenses to date constitute the most comprehensive software-based Spectre defenses of any widely deployed sandboxed execution environment (i.e., compared to the JVM [83], WASM [35, 120], and JavaScript [65, 79, 97]), their performance [60] and expressiveness [44] is limited.

Spectre-STL defenses primarily cause performance issues because they require a speculation barrier after most memory stores (e.g., pointer spills and stack initialization) [60]. Spectre-PHT defenses, on the other hand, primarily cause expressiveness issues (i.e., eBPF program rejections) because they use speculative path analysis (i.e., simulated mispredictions, Figure 1) to reject programs that execute architecturally forbidden operations under speculative execution (e.g., bit operations on registers containing pointers). Finally, the verifier uses masking to prevent out-of-bounds accesses in the presence of speculative scalar confusion [44].

3 Threat Model

Driven by unprivileged eBPF's numerous use cases [32, 34, 37, 41, 43, 55, 59, 103, 109] and without loss of generality, we consider unprivileged local users who can load eBPF programs into the kernel. Regarding the hardware, we consider all transient execution attacks for which prior work has shown speculative breakouts from eBPF, i.e., Spectre-PHT (v1), -BTB (v2), -STL (v4), -SLS and the corresponding half-Spectre variants. We thereby include all attacks considered by upstream eBPF, VeriFence, HIVE, BeeBox, and "Half Spectre, Full Exploit" [36]. For eBPF, existing defenses prevent Spectre-BTB and -SLS with negligible overhead, therefore, we focus on Spectre-PHT and Spectre-STL in the verifier. We

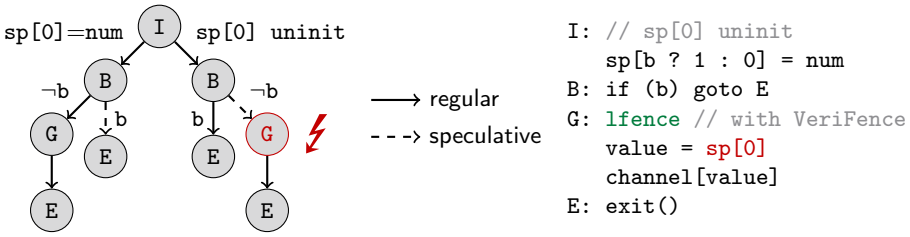

Fig. 1. Branch-based speculative type confusion in eBPF. Both Linux 6.16 and VeriFence analyze (architecturally impossible) speculative paths to prevent exploits. Linux 6.16 rejects the whole program upon encountering the attempted speculative read (`sp[0]` if stack slot is uninitialized and contains stale kernel data), whereas VeriFence inserts a speculation barrier (`lfence` on x64) to resolve the issue.

assume that CPUs may speculate on any branch or store instruction (we do not make assumptions about the CPU predictors). For side channels, we assume that any non-CT instruction constitutes a potential side channel leak that Firefly must prevent.

Like BeeBox, VeriFence, and upstream eBPF, we assume there are no bugs in the eBPF verifier or JIT engine. Although implementation bugs are also a relevant issue [72], they are the subject of numerous orthogonal related works [61,77, 84,85,91,100,105,106,115,116,122]. We assume that the kernel invokes the eBPF program, even speculatively, only with correctly typed and initialized arguments. Upstream Linux ensures this by applying transient execution attack defenses to the trusted kernel source code. Similarly, we assume kernel developers harden speculation triggers in helper C source code using the appropriate kernel abstractions (e.g., `array_index_nospec()` [98]). These are already inserted semi-automatically using specialized static analyzers such as Coverity [48,49]. Finally, we assume kernel developers check speculation triggers in inlined helper functions and harden them when required to ensure maximum performance (e.g., as done for map look-ups [104]). Upstream eBPF also makes all of these assumptions.

4 Problem Statement

Existing solutions to prevent eBPF-based transient execution attacks either incur performance penalties, lack portability, or complicate eBPF's integration into the kernel by requiring a separate address space. Table 1 summarizes these aspects for Firefly and related works.

4.1 Performance Penalties

All approaches that prevent transient execution attacks from eBPF incur significant overhead in typical eBPF use cases. For BeeBox and HIVE, this is because they introduce domain-switching overheads and have unnecessary overlap with hardware defenses. For upstream eBPF and VeriFence, this is because they suffer

Table 1. Comparison of eBPF transient execution attack defenses [44,60,127] regarding Portability (Sec. 4.2); Dynamic Sharing, Fine-Grained Isolation, and Abstraction Reuse (Sec. 4.3); as well as Low Invocation Overhead and eBPF execution-time overhead (Sec. 4.1). For HIVE, some aspects are unclear (⌣) because it is closed source.

	Port.	Dyn. Shar.	Fine-Gr. Isolation	Abstr. Reuse	Low Invoc. Overhead	eBPF Overhead x64	eBPF Overhead ARM64
HIVE	×[a]	⌣	×	⌣	×	×	5 % to 165 %[BT]
BeeBox-RB	✓[b]	×	×	×	✓	20 %[K]	⌣[c]
BeeBox-RC	✓[b]	✓	×	×	×	71 %[K]	⌣[c]
Linux 6.16	✓	✓	✓	✓	×	×	×
VeriFence	✓	✓	✓	✓	×	145 %[KL]	10 %[KL]
Firefly	✓	✓	✓	✓	✓	25 %[KL]	6 %[KL]

[a] MPK-based design for x64 discussed
[b] ARM64 and others not implemented
[c] Similar ARM64 and x64 overhead
[KL] Katran and Loxilb (Sec. 7.5, 7.6)
[K] Katran (Sec. 7.5 and [60])
[BT] BCC and Tracee [127]

from false positives in their static analysis and therefore insert excessive speculation barriers into the program.

Switching overhead between eBPF and the kernel. From a system's perspective, eBPF complements user space by enabling short-running user-defined computations (e.g., policy decisions) in the kernel without the expense of a costly switch to user space. As a result, most eBPF programs are small and short-running. For example, the 171 real-world application programs analyzed by Gerhorst et al. have a mean program size of 46 instructions [44], and the Katran load balancer benchmark also consumes at most 2750 CPU cycles. A low invocation overhead is thereby critical for eBPF.

It is therefore a problem that HIVE, BeeBox-RC (Reduced Copy), unprivileged upstream eBPF, and VeriFence all introduce domain-switching overhead. HIVE reports 165 % overhead for eBPF programs that consume less than 2000 CPU cycles, and only achieves 5 % if the program consumes at least 6000 CPU cycles. Similarly, the only BeeBox variant conceptually applicable to all eBPF program types (BeeBox-RC) must copy used objects into the eBPF address space before invocation, and therefore incurs an average overhead of 71 % for Katran. While privileged upstream eBPF has very low invocation overhead (similar to function calls), unprivileged upstream eBPF's and VeriFence's Spectre-STL defenses frequently require barriers after the stack initialization. As this usually occurs at program start, it indirectly incurs invocation overheads. Therefore, HIVE, BeeBox-RC, unprivileged upstream eBPF, and VeriFence incur domain-switching overheads that are problematic for eBPF's most common use cases.

Unnecessary overlap with hardware defenses. Neither BeeBox nor HIVE can eliminate overlap with hardware defenses because their designs are mono-

lithic. Their overhead is therefore the same on hardware that is only vulnerable to Spectre-PHT. While verification-based isolation can avoid duplicating Spectre-STL defenses on ARM64, where Speculative Store Bypass Disable (SSBD) is active by default in kernel mode [22, 129, 130], the performance benefit of this optimization in eBPF to date remains unclear.

Excessive overhead from speculation barriers. VeriFence and upstream eBPF insert speculation barriers on demand whenever their static analysis detects a potential Spectre-PHT or Spectre-STL gadget.

For Spectre-PHT, upstream eBPF analyzes the speculative execution paths triggered by branches and rejects the program if any architecturally forbidden behavior (e.g., type errors) occurs on a speculative path. Although this prevents exploits, it has the downside of causing many false-positive program rejections that are extremely tedious for application developers to resolve [44]. VeriFence avoids rejections by automatically adding a barrier, but it does not consider performance and introduces many unnecessary speculation barriers.

To prevent Spectre-STL, VeriFence and upstream eBPF insert a barrier after each type-/memory-safety–critical store instruction which implies that the verifier must add a barrier after each stack-slot initialization and pointer spill. Low-overhead Spectre-STL defenses are a well-recognized research challenge [14, 79].

4.2 Missing Portability

Upstream eBPF supports 12 architectures and implements transient execution attack defenses for three of them (x64, ARM64, and PowerPC). It is therefore impractical if defenses rely on non-portable hardware features. Especially, HIVE is not portable to any architecture other than ARM64 because it relies on Pointer Authentication Codes (PAC) and Load/Store Unprivileged (LSU) instructions for isolation [53]. BeeBox and VeriFence conceptually support architectures besides x64, but the authors neither evaluate their performance nor demonstrate portability in their implementations (both support only x64).

4.3 Dependence on Address-Space Separation

To allow fine-grained, low-overhead dynamic memory sharing and enable the reuse of kernel abstractions in helper functions, upstream eBPF and the kernel share the same address space. In contrast, BeeBox and HIVE separate the eBPF address space from the kernel address space to prevent Spectre-PHT and Spectre-STL. However, this separation inhibits both memory and abstraction sharing, as they only support coarse-grained static isolation and must duplicate kernel abstractions for use with eBPF pointers.

Static sharing. eBPF supports 32 program types in upstream Linux. Low-overhead defenses must therefore be generic and must not rely on optimizations that work only for specific programs. This problem primarily affects BeeBox,

whose most performant variant, BeeBox-RB (Ring-Buffer no-copy), has 20 % overhead for Katran but requires the kernel to statically select the target eBPF program when it allocates the memory to be shared with that program (static sharing). Only network driver programs can use this optimization [60], thereby severely limiting BeeBox-RB's use.

Jin et al. [60] present multiple variants to mitigate this limitation of BeeBox-RB, but these instead impose copying-overheads or excessive register pressure. First, BeeBox-RC implements dynamic sharing via invocation-time object copying across address spaces, but incurs 71 % overhead for Katran. Second, BeeBox-CP (Clean Pointers) avoids copying and supports dynamic sharing, but in turn requires the JIT to statically reserve one hardware register for every region shared with the program. Such static reservations are impractical for anything but trivial eBPF programs because of the register pressure they impose. While Jin et al. evaluate it manually for small socket filter programs, its overhead for complex real-world programs (e.g., Katran) remains unclear. In summary, none of BeeBox's variants supports low-overhead dynamic sharing.

Coarse isolation granularity. Because BeeBox and HIVE rely on address-space separation for isolation, they can only isolate data at coarse granularity. HIVE only supports fine-grained isolation by relying on Pointer Authentication Codes (PAC) and exception-based runtime patching, but this incurs significant initialization overhead and is not portable. For low-overhead sharing, HIVE and BeeBox-RB therefore require the kernel to place whole objects (e.g., C `struct`) in the correct eBPF program's address space. Consequently, kernel developers must remove all sensitive data from these objects, which induces additional indirections or copies.

Adapting the kernel to support the coarse isolation granularity required for BeeBox and HIVE is not trivial. This is, for example, clear from the fact that BeeBox supports only a small subset of eBPF interfaces while still changing 4800 kernel source lines. Even then, it only supports 5 out of 32 map types, 36 out of 211 helpers, and 2 out of 32 program types. A feature-complete BeeBox implementation would therefore lead to a significant increase in kernel complexity.

Abstraction reuse. Separate address spaces inhibit the reuse of kernel code and abstractions for eBPF. This is because developers must duplicate pointer-based kernel data structures (e.g., linked lists) for use in eBPF's kernel helper functions (e.g., BeeBox Listing 2 [60]). Furthermore, the system must convert eBPF pointers to kernel pointers and vice versa when switching between domains (as user pointers require special treatment in system calls). The latter affects 190 out of 211 of eBPF's kernel helper functions, which all receive pointer arguments from the eBPF program. These issues replicate the function-coloring problem from asynchronous programming languages [86]. Split address spaces risk further complicating the eBPF subsystem's implementation, but BeeBox and HIVE fail to address this concern because their implementations cover only a small subset of upstream eBPF's features.

5 Firefly Design

Based on the preceding problem analysis, Firefly follows three guiding principles:

Performance: Firefly offers high performance on ARM64 and x64, as our evaluation with the Katran and Loxilb real-world benchmarks shows.

Portability: Firefly uses speculation barriers and a portable constant-time–based leakage model that still hides the physical memory layout. It does not require non-portable hardware features such as ARM64 Pointer Authentication Codes, Load/Store Unprivileged instructions, x64 Memory Protection Keys, additional reserved registers, or information about the CPU caches and predictors.

Sharing: Firefly supports fine-grained, dynamic isolation and native kernel memory access using accurate verification-based isolation. Unlike related works, it thereby does not restrict memory sharing between eBPF and the kernel or prevent direct kernel code reuse in kernel helper functions.

Our work shows that a portable verification- and barrier-based design can achieve all three properties. For this, Firefly combines an accurate speculation- and hardware-aware verifier with compiler and interface refinements that make this approach scalable.

5.1 Unified Speculation-Aware Verification

Firefly's unified speculation-aware verification is the core mechanism that enables high performance. It addresses both Spectre-PHT and Spectre-STL in a portable manner while keeping the analysis scalable. This section gives an overview of the design, with details on the corresponding abstract values, instruction classes, and path-exploration rules deferred to our artifact's documentation.

Within the verifier's framework, Firefly adopts a constant-time–based [27,28] classification of speculative operations. We select speculative constant-time over Speculative Non-Interference (SNI) [39,47,90] because the verifier can efficiently enforce it using the type information it already maintains [15, 102, 112]. Firefly treats a speculative state as problematic only when a secret influences control flow, the address of a memory access, or the timing of instructions. In contrast, a speculative load may move a secret without immediately leaking it, provided the subsequent code uses the loaded value only in CT operations. Firefly treats loads as CT with respect to the loaded value and treats simple ALU instructions as CT with respect to their operands (these do not include multiplication, division, and modulo). Firefly therefore still prevents secret-dependent timing and cache contents, but avoids barriers for speculative states that remain confined to CT processing.

Regarding the Spectre-PHT example in Figure 1, Firefly adds a barrier before `channel[value]` because this is a non-CT operation, and `value` is a kernel secret (i.e., stale kernel data read from the uninitialized stack slot). For this gadget, Firefly therefore also prevents the exploit. In contrast, if `channel[value]`

is removed, VeriFence still adds an unnecessary barrier before `sp[0]`, whereas Firefly avoids it.

While the CT leakage model prevents secret-dependent memory offsets, this is not sufficient to protect the physical memory layout in the presence of the kernel's direct map (`page_offset_base`). However, disclosing the mapping from virtual to physical addresses to unprivileged users allows for predictable Rowhammer attacks [36]. For speculative out-of-bounds memory accesses such as `packet[size + 1]`, Firefly therefore only allows reads through the direct map if the access remains within the MMU page and DRAM row (at least 2 KiB [46,80,89,92,108]) of the object's allocation. This is possible because the verifier knows the object's type, and therefore its alignment-requirements, at verification-time. In the direct map, the virtual address alignment at 2 KiB granularity directly translates to physical address alignment [17], and thereby DRAM row alignment [108]. In particular, Firefly thereby does not assume a linear and monotonic mapping between physical and DRAM address spaces. If an out-of-bounds load is permitted, Firefly still tracks the loaded secret and subsequent code can only use it in constant time operations. In summary, the performance benefit of avoiding unnecessary barriers is thereby retained without admitting the DRAM layout leakage channel used by "Half Spectre, Full Exploit" [36].

To reduce the Spectre-STL defense overhead previously recognized by BeeBox and VeriFence, we apply speculative path analysis to Spectre-STL by simulating the bypass on architectures without in-kernel SSBD (i.e., x64 but not ARM64). This unified design simplifies the verifier and improves performance on x64. Figure 2 displays how Firefly applies speculation-aware path analysis to

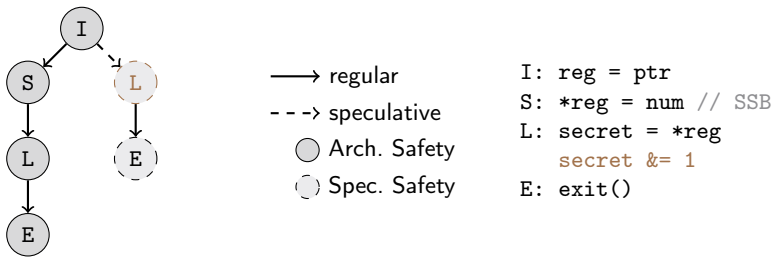

Fig. 2. Speculative store bypass (SSB) with Firefly's analysis. The program starts with stale secret kernel data stored at `*ptr`. Firefly accepts the non-problematic ALU operation using this secret in `L`, even though it does not satisfy architectural memory- and type-safety checks. Firefly only adds a barrier if non-CT branches or memory operations use `secret`.

Spectre-STL. On the speculative path, Firefly joins the pre- and post-store value of the affected slot into a single abstract value and gives each reload of that slot an independent identity (effectively bypassing the store in this example). The analysis then continues until it reaches a (possibly inserted) speculation barrier or program exit. This approach makes the verifier insert another barrier into the

subsequent code sequence only if the bypassed store causes the sequence to leak kernel secrets into side channels (thereby constituting a Spectre-STL gadget). However, in many cases, the bypassed store does not affect the safety of the following code, so that Firefly can avoid a barrier.

The main challenge with this approach is that it increases the number of execution paths the eBPF verifier must analyze, thereby negatively impacting startup time for systems that use complex unprivileged eBPF programs. To prevent analysis of non-trivial programs from becoming infeasible, Firefly applies upstream eBPF's path pruning logic [107] and limits the number of speculative instructions analyzed. When this configurable limit is reached, Firefly falls back to directly cutting speculative paths short using barriers. This fallback allows Firefly to successfully apply defenses to complex eBPF programs (e.g., Loxilb) by trading runtime overhead and verification time.

Both upstream eBPF and VeriFence use Spectre-PHT and Spectre-STL–specific bytecode instructions (`nospec_v1` and `nospec_v4`) for defenses against transient execution attacks [22, 44]. Firefly instead uses a single abstraction to stop both PHT- and STL-speculation (`nospec`). This allows for deduplication between PHT- and STL-barriers, which improves performance.

5.2 Whole-System Perspective

The verifier alone cannot achieve low-overhead defenses and instead requires a whole-system perspective. Firefly achieves this through hardware-aware verification as well as speculation-aware interfaces and compilation.

Hardware-aware verifier. Firefly allows the architecture layer to announce the vulnerabilities affecting the current CPU to the verifier, thereby further reducing the number of barriers on CPUs that do not require software defenses for specific Spectre variants. This in particular affects modern ARM64 CPUs (e.g., the Cortex-A76 microarchitecture used by cloud providers [16] and the Raspberry Pi 5 [96]) on which SSBD is active by default in kernel context. On these CPUs, Firefly therefore must not simulate store bypass and consequently inserts fewer barriers. We also consider dynamically enabling SSBD on x64 CPUs [15] when invoking eBPF code, however this is infeasible as the switch takes hundreds of cycles.

Speculation-aware interfaces. To reduce the number of Spectre-STL gadgets, Firefly uses pass-by-value for arguments to performance-sensitive helper functions whenever possible. For example, the common `bpf_map_lookup_elem()` helper used both for collection data structures (e.g., arrays and dictionaries) and inter-process communication, takes the collection key by reference. This, in turn, frequently creates a Spectre-STL gadget because the following key-lookup may speculatively use stale data (e.g., secret kernel pointers previously stored at the key's address) and leak them into a side channel. To resolve this, Firefly introduces an alternative to `bpf_map_lookup_elem()`, which takes 32- and 64-bit

keys by value. This covers the vast majority of applications because most array, dictionary, or communication-uses only require small keys.

Barrier-aware compilation. To reduce the performance impact of barriers in tight loops, Firefly configures the compiler that generates the bytecode (e.g., LLVM) to use more aggressive partial loop unrolling. As we show in Section 7.3, the associated increase in code size is negligible (e.g., 14 KiB for Loxilb).

6 Implementation

We implement Firefly for Linux 6.17 by modifying[6] approximately 1400 source lines and publish it under an open-source license.[5] We reimplement *fence or verify* [44] for Linux 6.16. From this reimplementation, we upstream 5 out of 7 resulting patches into 6.17[4] and implement Firefly based on this.[7] We upstream the unified speculation barrier abstraction (`nospec`) and the hardware-abstraction layer that allows architectures to request only the software defenses they actually require from the eBPF verifier (`bpf_jit_bypass_spec_v1/v4()`). We are not aware of any fundamental reasons that would prevent upstream Linux from merging the other Firefly patches in the future. To implement Firefly's Spectre-optimized bytecode, we add kernel helper function implementations that receive the map key by value (e.g., `map_lookup_elem_by_value()`) and double Clang's partial loop unrolling threshold (`-unroll-partial-threshold=300`). We finally upstream tests that ensure the verifier places speculation barriers appropriately for eBPF-based Spectre gadgets (e.g., for [23]).

7 Evaluation

We compare Firefly against VeriFence and BeeBox on both x64 and ARM64. As of April 2026, HIVE's source code is unavailable [117–119,127], therefore we cannot consider it in our evaluation. As benchmarks, we use real-world applications for which these related works have identified significant end-to-end application overhead due to eBPF's transient execution attack defenses.

7.1 Setup

We compare five configurations. (1) *No Defense*, which turns off only the eBPF-specific transient execution attack defenses while leaving other kernel defenses at their defaults (ARM64 in-kernel SSBD and x64 retpoline remain active). (2) *BeeBox-RB*, the fastest BeeBox variant with static memory sharing. (3) *VeriFence*, our Linux 6.17 reimplementation of the original version.[8] (4) *Firefly*, and

[6] We define modifications as the sum of SLOC insertions and deletions from `git diff`.
[7] We continue to refer to unprivileged eBPF in Linux 6.16 as *upstream eBPF*.
[8] We do this because Linux 6.5 is incompatible with the Loxilb version that Firefly requires. This version's performance only improves insignificantly upon the performance of the original VeriFence version.

(5) *Firefly No Def.*, which uses Firefly's speculation-aware interfaces and compilation without defenses and serves as Firefly's overhead baseline. VeriFence and BeeBox-RB overhead is still reported relative to *No Defense*.

We enable partial loop unrolling with the default x64 and ARM64 LLVM thresholds for all variants. Without defenses, the mean performance difference between Linux 6.17 and BeeBox's base Linux version (6.1.9) is 3 %. In our benchmarks, disabling in-kernel SSBD using the `ssbd=force-off` kernel command-line parameter improves performance by 4 % on average.

We run x64 performance measurements on an Intel server Xeon E5-4640 v2 at 2.2 GHz and ARM64 measurements on a Cortex-A76 system at 2.4 GHz [96]. Benchmarks execute in a KVM-based Debian 12 virtual machine, which is representative of common cloud deployments and aids reproducibility [16]. We validate the proof-of-concept exploits from Section 7.2 on an Intel Core i7-1185G7 and report the mean across 10 runs, with the first and third quartiles as error bars.

For real-world benchmarks, we select Katran and Loxilb because prior work reports substantial end-to-end overhead for transient execution attack defenses for these eBPF applications [44, 60, 106]. Other benchmarks from BeeBox and VeriFence show little or no mitigation overhead in their measurements and are therefore omitted (i.e., socket filters, `seccomp`, BCC tracers, and the Parca continuous profiler). Due to the many false-positive rejections before Linux 6.16, few production unprivileged eBPF applications exist today. We therefore follow prior work and enable the defenses for existing privileged benchmarks [21, 44] which preserves the relevant compiler, kernel-infrastructure, and verifier behavior. Our evaluation, therefore, accurately represents real-world unprivileged eBPF programs that users will develop in the future.

7.2 Security

This section summarizes Firefly's trusted computing base, our security argument, and presents experiments that verify our assumptions.

Trusted computing base. Because Firefly's Spectre-optimized eBPF bytecode is untrusted, only the verifier is relevant for security. Regarding implementation complexity, Firefly requires approximately 1400 source line modifications and is therefore unlikely to introduce new bugs into the verifier's implementation.

Systematization. As speculative constant-time is a formal security notion [14, 27, 102, 112], this section focuses on its extension to physical-layout information and application to eBPF. For this, we systematize Firefly's security properties within the formal framework developed by Cauligi et al. [28].

Regarding the execution model, relevant speculation triggers are branches and stores, and the eBPF program is invoked only with correctly typed inputs (even speculatively). We do not assume any specific CPU reorder buffer size to bound speculation, but only terminate speculative paths using speculation

barriers. We reference vendor documentation statements guaranteeing this for x64, ARM64, and PowerPC in our artifact's documentation.

Regarding the leakage model, we treat simple ALU operations as constant time, but not division, multiplication, or modulo [2, 4, 5]. In agreement with upstream eBPF, we model kernel function calls as opaque operations that are speculatively constant-time w.r.t. all kernel secrets (including the pointer, but not scalar, arguments they receive from eBPF programs). As in upstream eBPF, this is, for example, ensured using masking [104]. Inside the kernel, speculative constant-time does not directly capture the leakage of information about the physical memory layout. Firefly, therefore, refines it as follows. For accesses to eBPF-public memory regions (e.g., the stack), we only consider the offset relative to the object leaked, but still disallow crossing DRAM row boundaries beyond accessible allocations.

For the secrecy policy, we generally adopt unprivileged upstream eBPF's secrecy policy and identify kernel secrets based on existing value and type tracking. For network driver programs, we aim for a direct comparison with BeeBox-RB, which places the entire network interface controller (NIC) ring buffer in eBPF-public memory. We consequently also assume public packet buffers (i.e., `XDP_ZEROCOPY` UMEM chunks [30,81,111]) that are at least as large as the maximum transmission unit (MTU). For eBPF-public memory, the region offset is public, whereas the region's physical and virtual address are secret.

Experiments. Confirming that our implementation correctly enumerates canonical speculative paths from branches and stores, identifies non-CT operations, and tracks kernel secrets, we validate that Firefly prevents the three x64 Spectre-PHT and Spectre-STL exploits developed by Jin et al. [60].

7.3 Code Size

Katran and Loxilb use comparatively large eBPF programs. In contrast, for the 171 real-world application programs collected by Gerhorst et al., the mean program size is 46 instructions [44]. Katran uses one eBPF program with 2246 instructions whereas Loxilb loads 8 different programs with a total of 32651 bytecode instructions.

Firefly introduces no memory overhead at execution time except for the code size increase caused by partial loop unrolling and verifier-inserted defense instructions. For Loxilb, the undefended eBPF program code consumes 169 KiB after JIT compilation. Across architectures, Firefly increases this final code size by at most 14 KiB (i.e., 7 %). In contrast, BeeBox's changes to kernel memory layout increase whole-system memory usage by 4 MB to 12 MB [60].

7.4 Verification Time

Verification time is a secondary concern compared to runtime overheads, as it only affects application startup latency. Still, Firefly verifies even complex real-world programs in a reasonable time. For the most complex Loxilb program in

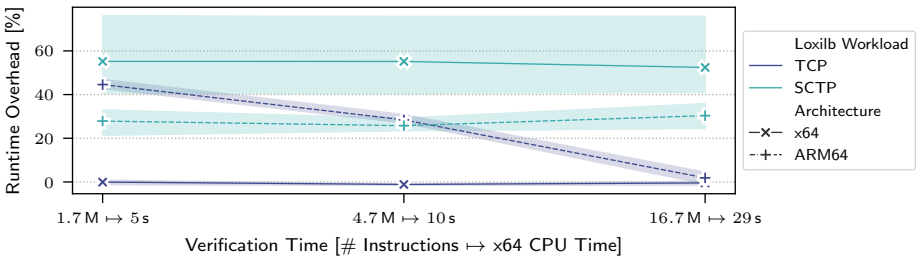

Fig. 3. Loxilb eBPF execution-time overhead for Firefly relative to *Firefly No Def.*. The resulting throughput reduction is lower. Firefly effectively trades verification time and eBPF execution time overhead.

our benchmarks, Firefly's analysis with defenses requires the verifier to process 16.7 M instructions (which consumes 29 s on our x64 CPU), therefore, long-running applications can amortize it. Without defenses, this same program requires the verifier to analyze 0.5 M instructions, which puts it within an order of magnitude of the architectural complexity limit (1 M instructions in Linux 6.16). On ARM64, Firefly's analysis takes 80 s because (1) the CPU is slower and (2) Spectre-PHT dominates the verification time. Out of the related works, only HIVE can eliminate verification, while BeeBox offers verification times similar to undefended eBPF (1.4 s on x64). VeriFence requires 10 s (4.1 M instructions) as it only analyzes Spectre-PHT paths without limiting speculative instruction analysis. In general, Loxilb's complexity constitutes a scenario that rarely occurs in production [44]. Katran, for example, requires 0.7 s on x64 (427 k instructions) with Firefly.

Firefly allows applications to trade shorter verification time for increased runtime overhead by lowering the configurable limit for speculative instruction analysis (set to 16 M by default). Figure 3 displays this trade-off for speculative instruction limits of 1 M, 4 M, and 16 M. Reducing this limit merely causes speculative paths to be cut off earlier with additional barriers, without compromising the system's security. This is because architectural paths (limited to 1 M in upstream Linux) are still always fully analyzed. Firefly still fences off all gadgets they contain. These architectural paths are the reason for the total number of analyzed instructions being 0.7 M instructions above the set limit in Figure 3. Developers can select an instruction limit suitable for their application based on their time budget and the verification performance numbers provided in this section. Alternatively, future work can run the full analysis in the background while letting the application already use the quick-analysis version.

7.5 Katran Benchmarks

Katran is a high-performance layer 4 load balancer developed by Meta. It includes a benchmarking utility that was also used to evaluate BeeBox.

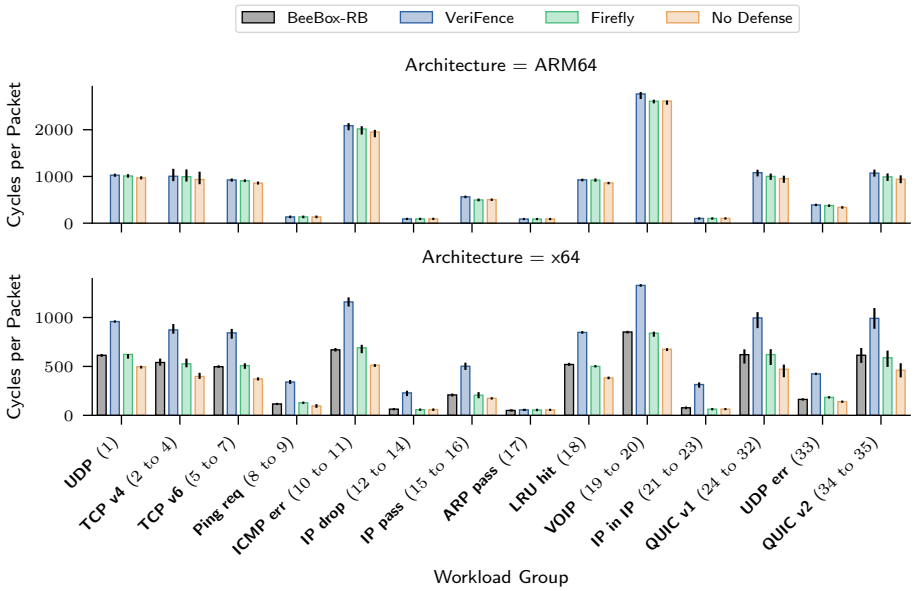

Fig. 4. Katran's XDP eBPF program benchmark for different CPU architectures.

Figure 4 displays the processing time per packet for all 35 different Katran workloads on x64 and ARM64. *Firefly No Def.*'s performance only differs insignificantly from the standard *No Defense* variant for Katran, therefore we omit it from the figure. To improve readability, we group the 35 workloads by type/execution time and list the corresponding Katran benchmark ID ranges for the 14 resulting groups. We calculate the geometric mean overhead for all defense approaches across the 35 workloads and display it in Figure 5.

With a geometric mean overhead of 6 % on ARM64 and 26 % on x64, Firefly offers high performance on both CPU architectures. On x64, Firefly offers performance similar to the most performant, but non-generic, BeeBox variant (BeeBox-RB), while VeriFence introduces a significant overhead of 151 %. BeeBox-RB has 26 % overhead on our Xeon E5-4640 v2 and 20 % on the Xeon W-2145 from the BeeBox paper. On ARM64, Firefly, like VeriFence, offers very low overhead of approximately 6 % because it can leverage the hardware's Spectre-STL defense and must only insert barriers against Spectre-PHT.

To summarize, Firefly offers performance similar to BeeBox-RB on x64, and significantly outperforms alternatives that support dynamic sharing (i.e., VeriFence). On ARM64, both Firefly and VeriFence have very low overhead because they can leverage hardware Spectre-STL defenses.

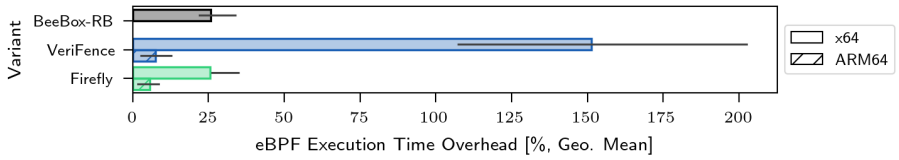

Fig. 5. Geometric mean overhead for the 35 Katran workloads.

7.6 Loxilb Throughput

Loxilb is a feature-rich load balancer and the primary benchmark from the VeriFence paper. It supports SCTP (e.g., as used by 5G networks [101]) and requires the eBPF `LPM_TRIE` map [24], which BeeBox's implementation does not support.

Figure 6 shows the `iperf3` TCP and SCTP throughput for two containers connected through the load balancer. As expected, TCP significantly outperforms SCTP because eBPF code calculates the CRC32 checksum for SCTP. For this reason, the undefended SCTP eBPF program on x64 consumes an average of 123.7 µs instead of TCP's 1.5 µs. This execution-time increase reduces throughput from 14.61 Gbit/s to 0.85 Gbit/s. Calculating the SCTP CRC32 checksum happens in a 4-line C loop that consumes the vast majority of the program's runtime. Any hardening instructions inserted into this loop have a significant effect on eBPF program runtime and, by extension, SCTP throughput. The right plot in Figure 6 shows the performance when manually adjusting the partial loop unroll count for the checksum-calculation loop. In contrast, the two left plots use the compiler-determined default value (i.e., 8 for VeriFence and 16 for Firefly).

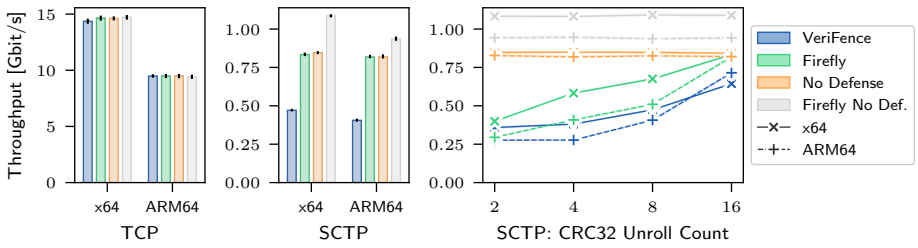

Fig. 6. TCP throughput for Loxilb v0.9.8.4 and the influence of partial CRC32 loop unrolling on SCTP throughput. Unrolling does not affect TCP performance. By default, Firefly's compiler selects an unroll count of 16, while VeriFence and *No Defense* default to a partial unroll count of 8.

Because Firefly's optimized bytecode can also increase performance without defenses, we benchmark this version separately as *Firefly No Def.* and always calculate Firefly's overhead relative to this improved version. We still use standard

No Defense as the baseline for VeriFence. Without defenses, Firefly's optimized bytecode improves SCTP throughput by up to 28 % while TCP throughput changes only insignificantly.

As displayed in the two left plots of Figure 6, Firefly significantly outperforms VeriFence in that SCTP throughput drops by 13 % to 23 % instead of 44 % to 50 %. For TCP, both have negligible overhead on x64 and ARM64. Our x64 VeriFence results thereby match those from [44], except for some expected differences due to the newer Loxilb version and a different CPU model.

For easier comparison between Loxilb and Katran, we measure the average Loxilb eBPF program runtime and calculate the respective defense's overhead. The x64 TCP/SCTP results confirm our conclusions from the Katran benchmark in that Firefly increases program runtime by 0 % to 51 % while VeriFence has 27 % to 98 % overhead. As for Katran, overhead for TCP on ARM64 is very low with both Firefly and VeriFence (at most 5 %). Only Firefly's SCTP overhead on ARM64 is higher than expected because any defense in the short checksum calculation loop incurs significant overhead. Still, with 30 % Firefly significantly outperforms VeriFence, which adds 154 % to the program runtime.

The right plot in Figure 6 displays how varying loop unroll counts affect SCTP overhead for Firefly and VeriFence. Whereas an undefended eBPF program's performance remains unaffected, loop unrolling significantly degrades performance with defenses because the inserted speculation barriers limit branch prediction. As shown in Figure 6, Firefly significantly outperforms VeriFence even with the unroll count fixed.

The results for Loxilb confirm our conclusions from the Katran benchmark, as Firefly significantly outperforms VeriFence in both end-to-end throughput and eBPF program execution time. They further demonstrate that Firefly supports complex eBPF programs and show the benefit of partial loop unrolling.

7.7 Evaluation Summary

Our evaluation answers the paper's main question in the affirmative: transient execution attack defenses for in-kernel sandboxes do not require address-space separation or architecture-specific hardware. Across all 37 tested Loxilb and Katran workloads, Firefly's geometric mean eBPF program execution time overhead is 25 % on x64 and 6 % on ARM64. It thus matches the most high-performance, but architecture-specific, state-of-the-art solutions (i.e., VeriFence on ARM64 and BeeBox-RB on x64) without relying on non-portable hardware features or static memory sharing.

8 Related Work

Transient execution attack defenses. Upstream eBPF and VeriFence use portable and non-intrusive verification-based isolation but suffer from high overhead, whereas BeeBox and HIVE depend on address-space isolation, which introduces portability and integration concerns. We compared VeriFence [44], Bee-

Box [60], and HIVE [127] to our approach in the problem statement and evaluation. Furthermore, static analysis tools focus on cryptographic user space software [28], but no existing work applies them to an in-kernel sandbox in a portable and scalable manner while still protecting the DRAM memory layout. For example, these tools assume impractically small speculation windows (Spectector [47], Pitchfork [27]), simplistic memory models [47], or focus on ahead-of-time compilers with a much larger trusted computing base and different application code properties than in-kernel sandboxes (Serberus [82], Blade [112], Jasmin [18]). None of them therefore answers whether verifier-based defenses for eBPF are viable from a systems perspective.

Runtime checks. SafeBPF [72] and MOAT [75] use SFI-, MTE-, and MPK-based isolation, but they do not consider transient execution attacks and raise portability and compatibility issues through coarse-grained isolation and slower dynamic sharing. Similar to Firefly, AEE [106] recognizes these limitations. It embeds runtime checks into eBPF programs to ensure the correctness of verifier abstractions, but does not consider transient execution. Future work could extend AEE to Firefly's unified speculation-aware verification.

Testing. Our upstreamed patches extend upstream eBPF's test suite to check that the verifier places speculation barriers correctly. Fuzz testing of eBPF is an active research area [77,91,105]. It can target speculative execution, and thereby Firefly, if speculation is traceable via performance counters [12, 78].

Formal methods. Multiple related works develop formalizations and soundness proofs for the eBPF verifier's C implementation [100, 115, 116]. As Firefly's design is already based on formal security notions, proving its implementation, or formalization thereof, secure is a natural subject of future research. Alternatively, Proof-Carrying Code (PCC) promises to reduce the eBPF verifier's complexity, and by extension its potential for bugs [61, 84, 85, 122]. Such verifiers can integrate Firefly's approach with a small trusted computing base.

9 Conclusion

This work presented Firefly and, with it, a concrete answer to an open systems question for in-kernel sandboxes. Prior work suggested that low-overhead defenses against transient execution attacks must sacrifice either portability or address-space sharing. Firefly shows that this trade-off is not fundamental. By removing the bottlenecks of verification-based defenses, it achieves a shared-address-space design that matches the best-performing prior designs on established real-world benchmarks. Firefly thereby paves the way towards portable, efficient, and securely sandboxed execution of unprivileged eBPF.

Acknowledgments. We thank Kumar Kartikeya Dwivedi, Eduard Zingerman, Sanidhya Kashyap, Alexei Starovoitov, Daniel Borkmann, Hari Bathini, Dustin Nguyen, and Maximilian Ott for giving their thoughtful feedback on the project and taking the time to review our kernel patches. We are grateful to the anonymous CCS'26 and RAID'26 reviewers for their insightful comments. This work was supported by the German Research Foundation (DFG) under projects number 539710462 (DOSS) and 502228341 (Memento), the German Federal Ministry of Education and Research (BMBF) under project number 16KIS2262 (SUSTAINET-inNOvAte), and a scholarship of the German Academic Exchange Service (DAAD).

Disclosure of Interests. The authors have no competing interests to declare.

References

1. Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115 (May 2018), https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html
2. DIT, Data Independent Timing - Arm Armv8-A Architecture Registers - Arm Documentation (Jun 2021), https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/DIT--Data-Independent-Timing
3. Refined Speculative Execution Terminology (Mar 2022), https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html
4. Data Operand Independent Timing Instructions (Feb 2023), https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html
5. Data Operand Independent Timing ISA Guidance (Feb 2023), https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html
6. Software techniques for managing speculation on AMD processors - revision 5.09.23 (2023), 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
7. Speculative processor vulnerability (updated December 20, 2023) - Arm security center (2023), https://web.archive.org/web/20240324073901/https://developer.arm.com/Arm%20Security%20Center/Speculative%20Processor%20Vulnerability
8. Affected processors: Guidance for security issues on Intel® processors (2024), 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
9. Discussion and FAQ for "Improving eBPF Complexity with a Hardware-backed Isolation Environment" - WANG, Zhe (Oct 2024), https://youtu.be/TGpteJoDog8?si=3kGNnLXqU2svxbPm&t=1247

10. Amaral, M., Chen, H., Chiba, T., Nakazawa, R., Choochotkaew, S., Lee, E.K., Eilam, T.: Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications. In: 2023 IEEE 16th International Conference on Cloud Computing (CLOUD). pp. 69–71 (Jul 2023). https://doi.org/10.1109/CLOUD60044.2023.00017, https://ieeexplore.ieee.org/document/10254956
11. AMD: CVE-2021-26341 - AMD CPUs May Transiently Execute Beyond Unconditional Direct Branch (2021), https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1026.html
12. Amit, N.: [RFC] x86: Speculative execution warnings (2019), https://lore.kernel.org/lkml/20190510192514.19301-1-namit@vmware.com/
13. ARM: Straight-line Speculation. Tech. rep. (2020), https://documentation-service.arm.com/static/6233364a8804d00769e9dbc1
14. Arranz-Olmos, S., Barthe, G., Blatter, L., Xie, X., Zhang, Z.: (Dis)Proving Spectre Security with Speculation-Passing Style (Oct 2025). https://doi.org/10.48550/arXiv.2510.11573, http://arxiv.org/abs/2510.11573
15. Arranz-Olmos, S., Barthe, G., Chuengsatiansup, C., Gregoire, B., Laporte, V., Oliveira, T., Schwabe, P., Yarom, Y., Zhang, Z.: Protecting Cryptographic Code Against Spectre-RSB: (and, in Fact, All Known Spectre Variants). In: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. pp. 933–948 (Mar 2025), https://doi.org/10.1145/3676641.3716015
16. AWS: AWS Graviton Technical Guide (Jun 2025), https://web.archive.org/web/20250614004026/https://github.com/aws/aws-graviton-getting-started/blob/main/README.md
17. Baoquan He: Line 150 in linux/arch/x86/mm/kaslr.c at master · torvalds/linux (2019), https://github.com/torvalds/linux/blob/1d51b370a0f8f642f4fc84c795fbedac0fcdbbd2/arch/x86/mm/kaslr.c#L150
18. Barthe, G., Cauligi, S., Grégoire, B., Koutsos, A., Liao, K., Oliveira, T., Priya, S., Rezk, T., Schwabe, P.: High-Assurance Cryptography in the Spectre Era. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1884–1901. IEEE, San Francisco, CA, USA (May 2021). https://doi.org/10.1109/SP40001.2021.00046, https://ieeexplore.ieee.org/document/9519434/
19. Begunkov, P.: io_uring: BPF controlled I/O (2021), https://lpc.events/event/11/contributions/901/
20. Borkmann, D.: BPF and Spectre: Mitigating transient execution attacks (2021), https://www.youtube.com/watch?v=6N30Yp5f9c4, eBPF Summit 2021
21. Borkmann, D., Horn, J., Starovoitov, A.: bpf: prevent out of bounds speculation on pointer arithmetic - kernel/git/torvalds/linux.git - Linux kernel source tree (Jan 2019), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=979d63d5
22. Borkmann, D., Krysiuk, P., Schlueter, B.: #f5e81d111750 - bpf: Introduce BPF nospec instruction for mitigating Spectre v4 - kernel/git/bpf/bpf.git - BPF kernel tree (2021), https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/commit/?id=f5e81d111750
23. Borkmann, D., Morrison, A., Kirzner, O., Schlüter, B., Krysiuk, P., Fastabend, J., Starovoitov, A.: Linux kernel source tree – bpf: Fix leakage under speculation on mispredicted branches (commit #918367) (2021), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9183671a
24. Brouer, J.D., Reimerink, D.: Map Type 'BPF_map_type_lpm_trie' - eBPF Docs (Oct 2025), https://web.archive.org/web/20251011033617/https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_LPM_TRIE/

25. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. p. 991 (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/bulck
26. Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., Von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A systematic evaluation of transient execution attacks and defenses. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 249–266 (2019), https://www.usenix.org/conference/usenixsecurity19/presentation/canella
27. Cauligi, S., Disselkoen, C., Gleissenthall, K.v., Tullsen, D., Stefan, D., Rezk, T., Barthe, G.: Constant-Time Foundations for the New Spectre Era. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA (Jun 2020). https://doi.org/10.1145/3385412.3385970
28. Cauligi, S., Disselkoen, C., Moghimi, D., Barthe, G., Stefan, D.: SoK: Practical Foundations for Software Spectre Defenses. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 666–680 (May 2022). https://doi.org/10.1109/SP46214.2022.9833707
29. Corbet, J.: Unprivileged bpf(). LWN.net (Oct 2015), https://lwn.net/Articles/660331/
30. Corbet, J.: Accelerating networking with AF_xdp. LWN.net (Apr 2018), https://lwn.net/Articles/750845/
31. Corbet, J.: Blocking straight-line speculation — eventually. LWN.net (Dec 2021), https://lwn.net/Articles/877845/
32. Corbet, J.: BPF and io_uring, two different ways. LWN.net (Nov 2025), https://lwn.net/Articles/1046950/
33. Daniel, L.A., Bognar, M., Noorman, J., Bardin, S., Rezk, T., Piessens, F.: ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. pp. 7161–7178 (2023), https://www.usenix.org/conference/usenixsecurity23/presentation/daniel
34. Daroc Alden: Filtering fanotify events with BPF (May 2025), https://lwn.net/Articles/1018493/
35. Dejaeghere, J., Gbadamosi, B., Pulls, T., Rochet, F.: Comparing security in eBPF and WebAssembly. In: Proceedings of the 1st Workshop on eBPF and Kernel Extensions (eBPF'23). pp. 35–41. ACM (2023). https://doi.org/10.1145/3609021.3609306
36. Di Dio, A., Hertogh, M., Giuffrida, C.: Half Spectre, Full Exploit: Hardening Rowhammer Attacks with Half-Spectre Gadgets. In: 2025 IEEE Symposium on Security and Privacy (SP). pp. 3583–3598 (May 2025). https://doi.org/10.1109/SP61157.2025.00207, https://ieeexplore.ieee.org/abstract/document/11023340, iSSN: 2375-1207
37. Edge, J.: Progress for unprivileged containers. LWN.net (Sep 2022), https://lwn.net/Articles/909627/
38. Edge, J.: Enhancing FineIBT. LWN.net (Oct 2025), https://lwn.net/Articles/1039633/
39. Fabian, X., Guarnieri, M., Patrignani, M.: Automatic detection of speculative execution combinations (2022). https://doi.org/10.48550/arXiv.2209.01179
40. Fabian, X., Patrignani, M., Guarnieri, M., Backes, M.: Do You Even Lift? Strengthening Compiler Security Guarantees against Spectre Attacks. Proc. ACM

Program. Lang. **9**(POPL), 31:893–31:922 (Jan 2025). https://doi.org/10.1145/3704867, https://dl.acm.org/doi/10.1145/3704867

41. Feng, C.: bpf: Remove the capability check for cgroup skb eBPF program - kernel/git/torvalds/linux.git - Linux kernel source tree (2017), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/kernel/bpf/syscall.c?id=80b7d81912d8
42. Gaidis, A.J., Moreira, J., Sun, K., Milburn, A., Atlidakis, V., Kemerlis, V.P.: FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 527–546. RAID '23 (Oct 2023). https://doi.org/10.1145/3607199.3607219, https://dl.acm.org/doi/10.1145/3607199.3607219
43. Gerhorst, L., Herzog, B., Reif, S., Schröder-Preikschat, W., Hönig, T.: Any-Call: Fast and Flexible System-Call Aggregation. In: Proceedings of the 11th Workshop on Programming Languages and Operating Systems. pp. 1–8. PLOS '21 (Oct 2021). https://doi.org/10.1145/3477113.3487267, https://doi.org/10.1145/3477113.3487267
44. Gerhorst, L., Herzog, H., Wägemann, P., Ott, M., Kapitza, R., Hönig, T.: VeriFence: Lightweight and precise Spectre defenses for untrusted Linux kernel extensions. In: Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '24) (2024). https://doi.org/10.1145/3678890.3678907
45. Ghimire, S., Bhurtel, N., Sahani, R., Jha, S.: eBPF-PATROL: Protective Agent for Threat Recognition and Overreach Limitation using eBPF in Containerized and Virtualized Environments (Nov 2025). https://doi.org/10.48550/arXiv.2511.18155, http://arxiv.org/abs/2511.18155
46. Ghose, S., Yaglikçi, A.G., Gupta, R., Lee, D., Kudrolli, K., Liu, W.X., Hassan, H., Chang, K.K., Chatterjee, N., Agrawal, A., O'Connor, M., Mutlu, O.: What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study. Proc. ACM Meas. Anal. Comput. Syst. **2**(3), 38:1–38:41 (Dec 2018). https://doi.org/10.1145/3224419, https://dl.acm.org/doi/10.1145/3224419
47. Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: Spectector: Principled Detection of Speculative Information Flows. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1–19 (May 2020). https://doi.org/10.1109/SP40000.2020.00011
48. Hagar Gamal Halim Hemdan: gpio: prevent potential speculation leaks in gpio_device_get_desc() - kernel/git/torvalds/linux.git - Linux kernel source tree (2024), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v6.18-rc6&id=d795848ecce24a75dfd46481aee066ae6fe39775
49. Hagar Gamal Halim Hemdan: vmci: prevent speculation leaks by sanitizing event in event_deliver() - kernel/git/torvalds/linux.git - Linux kernel source tree (Apr 2024), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=07ee18a0bc946b6b407942c88faed089e20f47d1&id=8003f00d895310d409b2bf9ef907c56b42a4e0f4
50. Heo, T.: [PATCHSET v3] sched: Implement BPF extensible scheduler class (2023), https://lore.kernel.org/all/20230317213333.2174969-1-tj@kernel.org/
51. Hertogh, M., Quakkelaar, D.J., Raymakers, T., Hari Sarma, M., Muench, M., Bos, H., van der Kouwe, E.: Rain: Transiently Leaking Data from Public Clouds

Using Old Vulnerabilities. In: 2026 IEEE Symposium on Security and Privacy (SP) (2026)
52. Hetterich, L., Bauer, M., Schwarz, M., Rossow, C.: Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security. pp. 217–230. ASIA CCS '24 (Jul 2024). https://doi.org/10.1145/3634737.3637662, https://dl.acm.org/doi/10.1145/3634737.3637662
53. Huang, K., Payer, M., Qian, Z., Sampson, J., Tan, G., Jaeger, T.: SoK: Challenges and Paths Toward Memory Safety for eBPF. In: 2025 IEEE Symposium on Security and Privacy (SP). pp. 848–866 (May 2025). https://doi.org/10.1109/SP61157.2025.00134, https://lightninghkm.github.io/files/eBPFSoK.pdf
54. Høiland-Jørgensen, T., Brouer, J.D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., Miller, D.: The eXpress data path: fast programmable packet processing in the operating system kernel. pp. 54–66. CoNEXT '18 (Dec 2018). https://doi.org/10.1145/3281411.3281443, https://doi.org/10.1145/3281411.3281443
55. Incus contributors: BPF token delegation (2025), https://linuxcontainers.org/incus/docs/main/explanation/bpf-tokens/
56. Intel: Managed runtime speculative execution side channel mitigations (2018), https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/runtime-speculative-side-channel-mitigations.html
57. Intel: Branch History Injection and Intra-mode Branch Target Injection (2025), https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html
58. Jia, J., Qin, R., Craun, M., Lukiyanov, E., Bansal, A., Phan, M., Le, M.V., Franke, H., Jamjoom, H., Xu, T., Williams, D.: Rex: Closing the language-verifier gap with safe and usable kernel extensions. pp. 325–342 (2025), https://www.usenix.org/conference/atc25/presentation/jia
59. Jia, J., Zhu, Y., Williams, D., Arcangeli, A., Canella, C., Franke, H., Feldman-Fitzthum, T., Skarlatos, D., Gruss, D., Xu, T.: Programmable System Call Security with eBPF (Feb 2023). https://doi.org/10.48550/arXiv.2302.10366, http://arxiv.org/abs/2302.10366
60. Jin, D., Gaidis, A.J., Kemerlis, V.P.: BeeBox: Hardening BPF against Transient Execution Attacks (2024), https://www.usenix.org/conference/usenixsecurity24/presentation/jin-di
61. Jin, D., Lavi, E., Jia, J., Lewis, R.Y., Vasilakis, N.: KLean: Extending Operating System Kernels with Lean. In: Proceedings of the 13th Workshop on Programming Languages and Operating Systems. pp. 118–126. PLOS '25 (Oct 2025). https://doi.org/10.1145/3764860.3768336, https://dl.acm.org/doi/10.1145/3764860.3768336
62. Jowett, A., Thaler, D., Saxena, A., Seal, S.: microsoft/ebpf-for-windows (v1.0.0-rc2) (2025), https://github.com/microsoft/ebpf-for-windows
63. Kernel Development Community: BPF Documentation — The Linux Kernel documentation (2025), https://www.kernel.org/doc/html/v6.16/bpf/index.html
64. Kim, D., Prast, R.: Triaging Real-time Security Threats with eBPF-powered Observability (2022), https://www.usenix.org/conference/srecon22americas/presentation/kim

65. Kim, J., van Schaik, S., Genkin, D., Yarom, Y.: iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 2038–2052. CCS '23 (Nov 2023). https://doi.org/10.1145/3576915.3616611, https://dl.acm.org/doi/10.1145/3576915.3616611
66. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1–19 (May 2019). https://doi.org/10.1109/SP.2019.00002
67. Krysiuk, P., Schlüter, B., Borkmann, D.: BPF and Spectre: Mitigating transient execution attacks (2022), https://popl22.sigplan.org/details/prisc-2022-papers/11/BPF-and-Spectre-Mitigating-transient-execution-attacks, keynote at PriSC'22
68. Kröning, M., Lankes, S., Klimt, J., Monti, A.: From Browser to Kernel: Exploring a Lightweight Sandboxed Approach for Unikernel Extensions. In: Proceedings of the 13th Workshop on Programming Languages and Operating Systems. pp. 127–135. PLOS '25 (Oct 2025). https://doi.org/10.1145/3764860.3768334, https://dl.acm.org/doi/10.1145/3764860.3768334
69. Lei, M.: Re: [PATCH v3 07/10] io_uring/bpf: implement struct_ops registration (Nov 2025), https://lore.kernel.org/all/aSRrundGeeIpaKmd@fedora/
70. Leitao, B., Poimboeuf, J.: CONFIG_mitigation_sls in bpf_jit_comp.c « net « x86 « arch - kernel/git/torvalds/linux.git - Linux kernel source tree (2023), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/net/bpf_jit_comp.c?h=v6.16#n680
71. Lian, Z., Li, Y., Chen, Z., Shan, S., Han, B., Su, Y.: eBPF-based Working Set Size Estimation in Memory Management. In: 2022 International Conference on Service Science (ICSS). pp. 188–195 (May 2022). https://doi.org/10.1109/ICSS55994.2022.00036, https://ieeexplore.ieee.org/abstract/document/9860164
72. Lim, S.Y., Prasad, T., Han, X., Pasquier, T.: SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. In: Proceedings of the 2024 on Cloud Computing Security Workshop. pp. 80–94. CCSW '24 (Nov 2024). https://doi.org/10.1145/3689938.3694781, https://dl.acm.org/doi/10.1145/3689938.3694781
73. Linux Foundation: eBPF Foundation (2025), https://ebpf.foundation/
74. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: Proceedings of the 27th USENIX Security Symposium (USENIX Security '18). pp. 973–990. USENIX (2018). https://doi.org/10.1145/3357033
75. Lu, H., Wang, S., Wu, Y., He, W., Zhang, F.: MOAT: Towards Safe BPF Kernel Extension. pp. 1153–1170 (2024), https://www.usenix.org/conference/usenixsecurity24/presentation/lu-hongyi
76. Lutomirski, A.: Re: [PATCH v2 bpf-next 1/4] bpf: unprivileged BPF access via /dev/bpf [LWN.net] (2019), https://lore.kernel.org/netdev/CALCETrWBWbNFJvsTCeUchu3BZJ3SH3dvtXLUB2EhnPrzFfsLNA@mail.gmail.com/
77. Lyu, T., Dwivedi, K.K., Bourgeat, T., Payer, M., Xu, M., Kashyap, S.: eBPF Misbehavior Detection: Fuzzing with a Specification-Based Oracle. In: Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles. pp. 701–718. SOSP '25 (Oct 2025). https://doi.org/10.1145/3731569.3764797, https://dl.acm.org/doi/10.1145/3731569.3764797

78. Mambretti, A., Neugschwandtner, M., Sorniotti, A., Kirda, E., Robertson, W., Kurmus, A.: Speculator: a tool to analyze speculative execution attacks and mitigations. In: Proceedings of the 35th Annual Computer Security Applications Conference. pp. 747–761. ACSAC '19 (Dec 2019). https://doi.org/10.1145/3359789.3359837
79. Mcilroy, R., Sevcik, J., Tebbi, T., Titzer, B.L., Verwaest, T.: Spectre is here to stay: An analysis of side-channels and speculative execution (Feb 2019). https://doi.org/10.48550/arXiv.1902.05178
80. Meyer, D., Jattke, P., Marazzi, M., Qazi, S., Moghimi, D., Razavi, K.: Phoenix: Rowhammer Attacks on DDR5 with Self-Correcting Synchronization. In: 2026 IEEE Symposium on Security and Privacy (SP) (2026)
81. Molè, M., Shahinfar, F., Tranquillo, F.M., Zoni, D., Panda, A., Antichi, G.: Performance Implications at the Intersection of AF_xdp and Programmable NICs. In: Proceedings of the 3rd Workshop on eBPF and Kernel Extensions. pp. 1–7. eBPF '25 (Sep 2025). https://doi.org/10.1145/3748355.3748359, https://dl.acm.org/doi/10.1145/3748355.3748359
82. Mosier, N., Nemati, H., Mitchell, J.C., Trippel, C.: Serberus: Protecting Cryptographic Code from Spectres at Compile-Time. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 4200–4219 (May 2024). https://doi.org/10.1109/SP54263.2024.00048, https://ieeexplore.ieee.org/abstract/document/10646752, iSSN: 2375-1207
83. Naseredini, A., Gast, S., Schwarzl, M., Bernardo, P.M.S., Smajic, A., Canella, C., Berger, M., Gruss, D.: Systematic Analysis of Programming Languages and Their Execution Environments for Spectre Attacks. pp. 48–59 (2022). https://doi.org/10.5220/0010779300003120
84. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97 (1997). https://doi.org/10.1145/263699.263712
85. Nelson, L., Wang, X., Torlak, E.: A proof-carrying approach to building correct and flexible in-kernel verifiers (2021), https://www.lukenels.net/slides/2021-09-23-lpc21.pdf, linux Plumbers Conference (LPC'21)
86. Nystrom, B.: What Color is Your Function? – journal.stuffwithstuff.com (2015), https://web.archive.org/web/20250929150146/https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/
87. Oleksenko, O., Fetzer, C., Köpf, B., Silberstein, M.: Revizor: Testing black-box CPUs against speculation contracts. In: ASPLOS. pp. 226–239 (2022). https://doi.org/10.1145/3503222.3507729
88. Osiński, T.: p4c-ubpf: a New Back-end for the P4 Compiler (Jun 2020), https://opennetworking.org/news-and-events/blog/p4c-ubpf-a-new-back-end-for-the-p4-compiler/
89. Patel, M., Kim, J.S., Mutlu, O.: The reach profiler (REAPER): Enabling the mitigation of DRAM retention failures via profiling at aggressive conditions. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). pp. 255–268 (Jun 2017). https://doi.org/10.1145/3079856.3080242, https://ieeexplore.ieee.org/document/8192476
90. Patrignani, M., Guarnieri, M.: Exorcising spectres with secure compilers. In: Proceedings of the 28th SIGSAC Conference on Computer and Communications Security (CCS'21). pp. 445–461. ACM (2021). https://doi.org/10.1145/3460120.3484534

91. Peng, C., Jiang, M., Wu, L., Zhou, Y.: Toss a Fault to BpfChecker: Revealing Implementation Flaws for eBPF runtimes with Differential Fuzzing. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 3928–3942. CCS '24 (Dec 2024). https://doi.org/10.1145/3658644.3690237, https://dl.acm.org/doi/10.1145/3658644.3690237
92. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. pp. 565–581 (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl
93. Prashar, V., Kumar, V.: Kepler (Kubernetes-based Efficient Power Level Exporter) - v0.11.3 (git-e9c8036) (Nov 2025), https://github.com/sustainable-computing-io/kepler
94. Qiao, F., Fang, Y., Cidon, A.: Energy-Aware Process Scheduling in Linux. SIGENERGY Energy Inform. Rev. **4**(5), 91–97 (Apr 2025). https://doi.org/10.1145/3727200.3727214, https://dl.acm.org/doi/10.1145/3727200.3727214
95. Raffin, G., Trystram, D.: Dissecting the Software-Based Measurement of CPU Energy Consumption: A Comparative Analysis. IEEE Transactions on Parallel and Distributed Systems **36**(1), 96–107 (Jan 2025). https://doi.org/10.1109/TPDS.2024.3492336, https://ieeexplore.ieee.org/abstract/document/10746340
96. Raspberry Pi Ltd: Raspberry Pi 5 16GB with Broadcom BCM2712 (Jan 2025), https://web.archive.org/web/20250903042708/https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf
97. Reis, C., Moshchuk, A., Oskov, N.: Site Isolation: Process Separation for Web Sites within the Browser. pp. 1661–1678 (2019), https://www.usenix.org/conference/usenixsecurity19/presentation/reis
98. Rutland, M.: Speculation — The Linux Kernel documentation (v6.16) (2018), https://www.kernel.org/doc/html/v6.16/staging/speculation.html
99. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross–privilege–boundary data sampling. In: Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19). pp. 753–768. ACM (2019). https://doi.org/10.1145/3319535.3354252
100. Shachnai, M., Vishwanathan, H., Narayana, S., Nagarakatte, S.: Fixing Latent Unsound Abstract Operators in the eBPF Verifier of the Linux Kernel. In: Giacobazzi, R., Gorla, A. (eds.) Static Analysis, vol. 14995, pp. 386–406. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-74776-2_15, https://link.springer.com/10.1007/978-3-031-74776-2_15, series Title: Lecture Notes in Computer Science
101. Sharma, B., Vittal, S., Franklin, A.A.: FlexCore: Leveraging XDP-SCTP for Scalable and Resilient Network Slice Service in Future 5G Core. In: Proceedings of the 7th Asia-Pacific Workshop on Networking. pp. 61–66. APNet '23 (Sep 2023). https://doi.org/10.1145/3600061.3600073, https://dl.acm.org/doi/10.1145/3600061.3600073
102. Shivakumar, B.A., Barthe, G., Gregoire, B., Laporte, V., Oliveira, T., Priya, S., Schwabe, P., Tabary-Maujean, L.: Typing High-Speed Cryptography against Spectre v1. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 1094–1111 (May 2023). https://doi.org/10.1109/SP46215.2023.10179418, https://ieeexplore.ieee.org/document/10179418
103. Starovoitov, A.: bpf: enable non-root eBPF programs - kernel/git/torvalds/linux.git - Linux kernel source tree (2015), https:

//git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/kernel/bpf/syscall.c?id=1be7f75d1668
104. Starovoitov, A., Fastabend, J., Borkmann, D.: bpf: prevent out-of-bounds speculation (Jan 2018), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b2157399cc98
105. Sun, H., Su, Z.: Validating the eBPF Verifier via State Embedding. pp. 615–628 (2024), https://www.usenix.org/conference/osdi24/presentation/sun-hao
106. Sun, H., Su, Z.: Approximation Enforced Execution of Untrusted Linux Kernel Extensions (2025), https://www.usenix.org/system/files/usenixsecurity25-sun-hao.pdf
107. Tardy, M., Chaignon, P.: Making Sense of State Pruning - Linux Plumbers Conference 2025 (2025), https://lpc.events/event/19/contributions/2162/
108. Tatar, A., Giuffrida, C., Bos, H., Razavi, K.: Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In: Bailey, M., Holz, T., Stamatogiannakis, M., Ioannidis, S. (eds.) Research in Attacks, Intrusions, and Defenses, vol. 11050, pp. 47–66. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-00470-5_3, http://link.springer.com/10.1007/978-3-030-00470-5_3, series Title: Lecture Notes in Computer Science
109. Tissoires, B.: HID-BPF — The Linux Kernel documentation (2023), https://www.kernel.org/doc/html/v6.16/hid/hid-bpf.html
110. Turner, P.: Retpoline: a software construct for preventing branch-target-injection (2025), https://support.google.com/faqs/answer/7625886
111. Töpel, B., Karlsson, M.: AF_xdp — The Linux Kernel documentation (2025), https://www.kernel.org/doc/html/v6.16/networking/af_xdp.html
112. Vassena, M., Disselkoen, C., Gleissenthall, K.v., Cauligi, S., Kıcı, R.G., Jhala, R., Tullsen, D., Stefan, D.: Automatically eliminating speculative leaks from cryptographic code with blade. Replication package for article: Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade **5**(POPL), 49:1–49:30 (Jan 2021). https://doi.org/10.1145/3434330, https://dl.acm.org/doi/10.1145/3434330
113. Vernet, D., Krishnan, S.: BPF/eBPF (bpf) (2025), https://datatracker.ietf.org/wg/bpf/about/
114. Vieira, M.A.M., Castanho, M.S., Pacífico, R.D.G., Santos, E.R.S., Júnior, E.P.M.C., Vieira, L.F.M.: Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. ACM Computing Surveys **53**(1), 16:1–16:36 (Feb 2020). https://doi.org/10.1145/3371038, https://doi.org/10.1145/3371038
115. Vishwanathan, H., Shachnai, M., Narayana, S., Nagarakatte, S.: Sound, precise, and fast abstract interpretation with tristate numbers. In: Proceedings of the 20th International Symposium on Code Generation and Optimization (CGO'22). pp. 254–265. IEEE (2022). https://doi.org/10.1109/CGO53902.2022.9741267
116. Vishwanathan, H., Shachnai, M., Narayana, S., Nagarakatte, S.: Verifying the Verifier: eBPF Range Analysis Verification. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, vol. 13966, pp. 226–251. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37709-9_12, https://link.springer.com/10.1007/978-3-031-37709-9_12, series Title: Lecture Notes in Computer Science
117. Wang, Z.: Improving eBPF Complexity with a Hardware-backed Isolation Environment (2024), https://lpc.events/event/18/contributions/1947/

118. Wang, Z., Zhang, P.P.: Research Update: Isolated Execution Environment for eBPF (Part 2) – eBPF (2025), https://ebpf.foundation/research-update-isolated-execution-environment-for-ebpf-part-2/
119. Wang, Z., Zhang, P.P.: Research Update: Isolated Execution Environment for eBPF – eBPF (2025), https://ebpf.foundation/research-update-isolated-execution-environment-for-ebpf/
120. Wasmtime: Security - Wasmtime (Feb 2024), https://docs.wasmtime.dev/security.html#spectre
121. Wiebing, S., Tron, A.d.F., Bos, H., Giuffrida, C.: InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2. pp. 577–594 (2024), https://www.usenix.org/conference/usenixsecurity24/presentation/wiebing
122. Wu, X., Feng, Y., Huang, T., Lu, X., Lin, S., Xie, L., Zhao, S., Cao, Q.: VEP: A Two-stage Verification Toolchain for Full eBPF Programmability. In: Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (2025), https://www.usenix.org/system/files/nsdi25-wu-xiwei.pdf
123. Xu Kuohai: [PATCH bpf-next] bpf: arm64: Fix panic due to missing BTI at indirect jump targets (2025), https://lore.kernel.org/bpf/20251127140318.3944249-1-xukuohai@huaweicloud.com/
124. Yang, J., Chen, L., Bai, J.: Redis automatic performance tuning based on eBPF. In: Proceedings of the 14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA'22). pp. 671–676. IEEE (2022). https://doi.org/10.1109/ICMTMA54903.2022.00139
125. Yutaro Hayakawa: eBPF Implementation for FreeBSD :: FreeBSD Presentations and Papers (2018), https://papers.freebsd.org/2018/bsdcan/hayakawa-ebpf_implementation_for_freebsd/
126. Zarkadas, I., Zussman, T., Carin, J., Jiang, S., Zhong, Y., Pfefferle, J., Franke, H., Yang, J., Kaffes, K., Stutsman, R., Cidon, A.: BPF-oF: Storage Function Pushdown Over the Network (Dec 2023). https://doi.org/10.48550/arXiv.2312.06808, http://arxiv.org/abs/2312.06808
127. Zhang, P., Wu, C., Meng, X., Zhang, Y., Peng, M., Zhang, S., Hu, B., Xie, M., Lai, Y., Kang, Y., Wang, Z.: HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 163–180 (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-peihua
128. Zhong, Y., Li, H., Wu, Y.J., Zarkadas, I., Tao, J., Mesterhazy, E., Makris, M., Yang, J., Tai, A., Stutsman, R., Cidon, A.: XRP: In-Kernel storage functions with eBPF. In: Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22). pp. 375–393 (Jul 2022), https://www.usenix.org/conference/osdi22/presentation/zhong
129. Zyngier, M.: arm64: Call ARCH_workaround_2 on transitions between EL0 and EL1 - kernel/git/bpf/bpf-next.git - BPF next kernel tree (2018), https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=8e2906245f1e3b0d027169d9f2e55ce0548cb96e
130. Zyngier, M.: [PATCH 00/14] arm64 SSBD (aka Spectre-v4) mitigation [LWN.net] (2018), https://lore.kernel.org/linux-kernel/20180522150648.28297-1-marc.zyngier@arm.com/