Verified Programs Can Party: Optimizing Kernel Extensions via Post-Verification Merging

Anonymous Author(s)
Submission ID: 112

Abstract

Operating system (OS) extensions are more popular than ever. For example, Linux BPF is marketed as a "superpower" that allows user programs to be downloaded into the kernel, verified to be safe and executed at kernel hook points. So, BPF extensions have high performance and are often placed at performance-critical paths for tracing and filtering.

However, although BPF extension programs execute in a shared kernel environment and are already individually verified, they are often executed independently in *chains*. We observe that the chain pattern has large performance overhead, due to indirect jumps penalized by security mitigations (*e.g.,* Spectre), loops, and memory accesses.

In this paper, we argue for a separation of concerns. We propose to decouple the execution of BPF extensions from their verification requirements—BPF extension programs can be collectively optimized, after each BPF extension program is individually verified and loaded into the *shared* kernel.

We present KFuse, a framework that *dynamically* and *automatically* merges chains of BPF programs by transforming indirect jumps into direct jumps, unrolling loops, and saving memory accesses, without loss of security or flexibility. KFuse can merge BPF programs that are (1) installed by multiple principals, (2) maintained to be modular and separate, (3) installed at different points of time, and (4) split into smaller, verifiable programs via BPF tail calls. KFuse demonstrates 85% performance improvement of BPF chain execution and 7% of application performance improvement over existing BPF use cases (`systemd`'s Seccomp BPF filters). It achieves more significant benefits for longer chains.

1 Introduction

Operating system (OS) extensions are more popular than ever. Linux BPF is marketed as a "superpower" that allows user programs to be downloaded into the kernel, verified to be safe and executed at kernel hook points. Currently, BPF extensions are used for system call security (*e.g.,* seccomp-BPF [22]), performance tracing (*e.g.,* tracepoints [26], bcc [2]), and network packet processing (*e.g.,* express data path (XDP) [28, 47]); many other use cases are proposed recently [2, 10, 12, 27, 28, 33–35, 37, 39–42, 48, 56, 60, 66–68, 71, 73–75]. BPF extensions have high performance, because BPF code is highly optimized [19] and BPF programs are executed entirely in the kernel. For example, sandboxing system call with seccomp BPF filters is reported to be 2× faster, compared with ptrace-based system call filtering [50]; using XDP [28, 47] to implement firewall rules improves network throughput by 11× compared to iptable-based implementations [20].

Currently, BPF extensions are individually verified by an in-kernel BPF verifier and then executed independently. A common execution pattern for BPF extensions is to execute a *chain* of independently-loaded BPF extensions that are attached at the same hook point. The chain pattern is a natural implementation choice to support BPF extensions that are (1) installed by multiple principals [6, 22, 26]; (2) installed at different points of time (*e.g.,* for temporally-specialized security policies [46, 57]), and (3) maintained to be modular and separate (for maintainability and debuggability) [16].

More fundamentally, due to the limited scalability of BPF program verification, the BPF verifier enforces each BPF extension to be small in size—each BPF program is limited to 4096 instructions with a maximum stack size of 512 bytes; therefore, large BPF features need to be split into small BPF programs, each of which can be verified in time [11].

However, we observe that the chain pattern has large performance overhead, due to indirect jumps penalized by security mitigations (*e.g.,* Spectre), loops, and memory accesses. The typical chain pattern is implemented by a sequence of *indirect jumps*. The kernel maintains an array of pointers of loaded BPF extensions which are called one by one via indirect jumps to these pointers. While indirect jumps were optimized in modern CPUs via branch prediction, recent security mitigations to speculative vulnerabilities [54] (Spectre, specifically Variant 2) incur significant overhead to indirect jumps, negatively affecting the execution of the chained BPF execution. For example, the *de facto* software mitigation, Retpoline [21], makes indirect jumps 13.3× slower [7, 64]. Besides loops, BPF extension chains can also be formed by tail calls; a BPF program can tail-call another BPF program, which takes a memory access and an indirect jump.

To make the matter worse, the overhead of BPF extension chains increases with the increase of the length of the chain. We have seen real-world BPF use cases with long BPF extension chains. For example, `systemd` [24] installs 19 seccomp BPF filters to its services. Our benchmark on Redis initialized by `systemd` shows that the chained BPF extensions can cause a 10% slowdown of Redis. We expect such long chain to be commonplace in future BPF use cases for fine-grained policies and richer features. The overhead of the chains would be untenable.

In this paper, we argue that execution and verification should be a separation of concerns. We propose to decouple the execution of BPF extensions from their verification requirements and limits—kernel extension programs can be collectively optimized, after each extension is individually verified and loaded into the *shared* kernel. We demonstrate that the decoupling could lead to new opportunities for performance optimizations, while maintaining the safety of the verified extension programs.

We present KFuse, a framework that dynamically and automatically optimizing chains of BPF extensions by transforming indirect jumps into direct jumps, unrolling loops, and saving memory accesses, without loss of security or flexibility. Specifically, KFuse merges a chain of BPF extensions into a fused BPF program by rewriting the return instructions into direct jump instructions and updating the jump offsets. For loop-based chains, KFuse updates the return values based on the original loop semantics (tail calls do not return). KFuse ensures that the fused BPF program maintains all the safety properties required by the BPF verifier. This can be proved by induction-based on the safety properties of the original BPF extension programs in the chain (ensured by the BPF verifier). Lastly, KFuse is fully transparent to user space. It achieves the transparency by maintaining the original verified BPF program and only performing the optimizations for execution. We integrate KFuse into three Linux kernel subsystems that adopt BPF: seccomp, tracepoint, and cgroup. We show that KFuse can be easily integrated in existing BPF use cases—it takes only 20, 28, and 34 lines of code changes for the three subsystems respectively.

We evaluate the effectiveness of KFuse in optimizing execution of BPF extension chains with real-world BPF use cases (the `systemd` BPF filters) as well as synthetic BPF extensions for scalability analysis. KFuse demonstrates 85% performance improvement of BPF chain execution and 7% of application performance improvement over existing BPF use cases (`systemd`'s seccomp BPF filters). When the chain is long (160 BPF filters), KFuse is able to improve NGINX performance by as much as 2.3 times.

This paper makes the following contributions:

- We propose a new perspective of optimizing a chain of verified BPF extensions collectively;
- We benchmark the overhead of executing BPF extension chains, based on which we identify the root causes and analyze their performance impacts;
- We present KFuse, an in-kernel framework for effectively optimizing BPF extension chains.

We will release all the code and dataset in the paper.

2 BPF Extension and The Chain Pattern

In this section, we introduce BPF extensions for the Linux kernel and identify how admitting extensions into the shared kernel creates a common chain pattern where multiple extensions are attached to the same hook.

2.1 BPF extensions

Linux supports kernel extensions implemented in the BPF (Berkeley Packet Filter) language [61]. BPF was originally used for network packet filters; later, it is adopted by many kernel subsystems, including security [13, 22], tracing [2, 3, 26], and process management [6]. Recent research has proposed a variety of BPF use cases, including storage [78], virtualization [32], hardware offloading [28, 51], and others [37, 70]. Note that in Linux, "BPF" stands for extended BPF (or *eBPF* in short). While some subsystems, such as seccomp, still use the classic BPF (cBPF) language, they are converted into the eBPF bytecode at the execution time. So, we focus our discussion on eBPF.

BPF has ten registers and a 512-byte stack. Instructions are 9-byte long and implement a general-purpose RISC instruction set. When an extension is loaded into the kernel, Linux compiles its BPF instructions "just in time" (JIT) into native instructions for better performance. The JIT compilation is optional. BPF programs maintain states via BPF maps. BPF maps are kernel objects that are in the form of key-value pairs, which can also be exposed to user space.

Every BPF program needs to pass an in-kernel BPF verifier; BPF programs that fail the verification will be rejected. The verifier ensures that the BPF extension program does not have unbounded loops, unreachable code, or out-of-bound jumps. So, a verified program is guaranteed to terminate, never jump to invalid locations and never access invalid out-of-bound memory [11]. To make sure the verification can be done in time, a BPF extension is limited to 4096 instructions.

In order to support more complex, modular extension programs, BPF includes a *tail call* mechanism that allows one program to call another one without returning. These programs are verified independently. The stack frame from the old program is unwound and reused. A program array that specifies available programs to tail call needs to be populated first. The program then reads the array and tail calls the target program.

In this paper, we focus on three Linux kernel subsystems that support BPF extensions, seccomp [22], tracepoint [26] and cgroup [6]. Seccomp uses BPF extensions for system call filters that restrict the system calls and their argument values a process can invoke. Tracepoint enables BPF extensions to observe the kernel states for profiling and debugging. Cgroup (control groups) limits the resource usage of processes and filters network traffic with BPF extensions.

2.2 Emergence of BPF chain patterns

A chain of BPF extensions is formed when multiple extensions are attached to the same kernel hook point. This is typically implemented in the form of loops. Figure 1 shows three simplified examples from seccomp (Figure 1a), tracepoint (Figure 1b) and cgroup (Figure 1c). Another way to form a chain is through tail calls—a BPF program (the caller)

```
for (; f; f = f->prev) {
    u32 cur_ret =
↪   bpf_prog_run(f->prog, sd);
    if (cur_ret < ret) {
        ret = cur_ret; *match = f;
    }
}
```

(a) Seccomp execution loop

```
u32 _ret = 1;
_item = &_array->items[0];
while ((_prog = _item->prog)) {
    _ret &= func(_prog, ctx);
    _item++;
}
return _ret;
```

(b) Tracepoint execution loop

```
while ((_prog =
↪   READ_ONCE(_item->prog))) {
    ...
    _ret &= func(_prog, ctx);
    ...
    _item++;
}
```

(c) Cgroup execution loop

Figure 1. Common patterns of executing BPF extensions: A chain of BPF programs are store in an array or a list and are executed in loops of indirect jumps.

can tail-call another BPF program (the callee), and the callee can tail-call other BPF programs.

We categorize four reasons of the chain pattern: (1) extensions are installed from different principals, (2) developers enforce modularity of kernel extensions, (3) extensions are installed at different points in time, and (4) extensions need to be verified independently.

Extensions installed by different principals. Extensions from different principals can not be combined together before being loaded to the kernel, because they may not trust or even be aware of each other. These extensions are verified independently and stored in an array of extensions, and, therefore, a chain is formed. Take seccomp-BPF as an example. Seccomp BPF filters can also be installed from different principals. In a cluster management system, host-specific seccomp BPF filters are installed [69]; the container runtime (*e.g.*, Docker) can install container-wide BPF filters; and applications can install application-specific BPF filters. All these filters are attached at the same location (the system call entry point). In tracepoint, multiple extensions can be attached to the same tracepoint. Processes from different principals collect performance statistics by installing their own tracepoint extensions. Another example is infrared (IR) decoding—decoding protocols are expressed as BPF extensions that can come from different principals.

Modularity of extensions. Modularity is a reason why developers maintain extensions to be single-purpose and separate. A systemd developer writes in the mailing list [16] regarding the usage of seccomp filters: *"Keeping the filters separate made things a lot easier and simpler to debug, and our log output and testing became much less of a black box. We know exactly what worked and what didn't, and our test validates each filter"*. Cgroup firewall rules are also easier to maintain, debug and reuse when these rules are simple. For instance, a common way to build a firewall is to have a rule that denies all packets and rules to allow specific traffic. Each rule is implemented as a BPF extension, which can be reused and maintained for different policies. One use case of the tail call is to separate big and complicated extensions into smaller and modular ones and chain them together.

Extensions installed at different points in time. Not all extension programs are known a priori so they can not be combined before being loaded. For example, *temporal system call specialization* [46, 57] is known for security benefits by installing filters at different execution phases of the application (*e.g.*, the initialization phase and the serving phase). Recent work shows that temporal system call specialization can reduce 51% more of the attack surface, in terms of the system call interface. Seccomp ensures that a BPF filter, once installed, cannot be removed during the process lifecycle. Therefore, temporal specialization leads to multiple seccomp filters to be installed at different points in time, forming a BPF program chain. Besides seccomp, tracepoint and cgroup also allow BPF extensions to be dynamically installed at different points in time, on demand.

Extensions that need to be verified individually. A more fundamental reason of the chain pattern comes from the limited scalability of BPF verification. The verification time depends on the size and complexity of the BPF program. To make sure that a user-supplied BPF extension can be verified in time, the verifier imposes limitations on the size and complexity of the BPF extensions, including the number of instructions, the number of jumps, the stack size, etc. Therefore, to pass the verification, a large, complex BPF extension need to be split into small-sized programs in a chain.

3 The Cost of Chaining Extensions

In this section, we benchmark the overhead of executing a BPF extension chain. The main overhead comes from the *indirect jumps* that are invoked to switch extensions on the chain and the *loops* that are used to iterate extensions.

3.1 Methodology

We design the benchmark for measuring the overhead of a chain of BPF executions. A chain can be formed when multiple extensions are attached to the same kernel hook point, or when a extension tail-calls other extensions. When multiple BPF extensions are attached to the same hook point, these extensions are executed in a loop (exempified in Figure 1).

```
void func1(){}
void func2(){}
void func3(){}
void (*funcs[3])() = {func1, func2,
↪  func3};
start = get_timestamp();
for(int i = 0; i < 3; i++) funcs[i]();
end = get_timestamp();
```

(a) Indirect jumps in a loop

```
void func1(){}
void func2(){}
void func3(){}
void (*fptr1)() = func1;
...
start = get_timestamp();
(*fptr1)(); (*fptr2)(); (*fptr3)();
end = get_timestamp();
```

(b) Indirect jumps with an unrolled loop

```
void func1(){}
void func2(){}
void func3(){}
start = get_timestamp();
func1();
func2();
func3();
end = get_timestamp();
```

(c) Direct jumps with an unrolled loop

Figure 2. Benchmarks for analyzing the cost of executing a chain of BPF extensions: (a) measures the cost of loops and indirect jumps. **(b)** measures the cost of indirect jumps. **(c)** measures the cost of direct jumps. Comparing **(a)** and **(b)** to obtain the cost of loops. Comparing **(b)** and **(c)** to obtain the cost of indirect jumps.

	Retpoline	No Retpoline
Loop + Indirect jumps (Fig. 2a)	47	9.3
Unrolled + Indirect jumps (Fig. 2b)	34.6	2.6
Unrolled + Direct jumps (Fig. 2c)	2.6	2.6

Table 1. Cycles needed to call a BPF extension: Indirect jumps incur 32 additional cycles with retpoline. Loops incur 12.4 additional cycles with retpoline and 6.7 without.

The loop iterates a program array and indirectly jumps to each item on the array.

We firstly establish the baseline and measure a chain of N extensions by calling indirect jumps to 3 different functions in a loop via function pointers, as shown in Figure 2a. Then we unroll the loop, as shown in Figure 2b, to obtain the loop cost by comparing the results from Figure 2a and Figure 2b. Finally, we convert indirect jumps to direct jumps, as shown in Figure 2c to obtain the indirect jump cost by comparing the results from Figure 2b and Figure 2c. A extension that uses tail calls needs to load the target program from a program array. This operation incurs memory access cost which is also part of the cost of executing extensions by a loop. Therefore, we reuse the benchmark and present the result in Section 3.2.3

We conduct all the benchmarking experiments on a single server with a 16 core Intel Xeon Silver 4110 CPU at 2.10GHz and 64 GB of memory. We use the `rdtsc` instruction to read the CPU cycles (*i.e.,* `get_timestamp`). We use the `cpuid` instruction to prevent instruction reordering.

3.2 Benchmark results

Table 1 summarizes our main results. In short, executing a function directly with unrolled loops takes 2.6 cycles; executing a function in loops with indirect jumps takes up to 47 cycles when retpoline is enabled, a 18.08× increase.

3.2.1 Indirect jump cost. The common pattern of executing a chain of BPF extensions is to execute *indirect jumps* to locations where these extensions are loaded. An indirect jump (as known as an indirect branch) is a jump instruction with an argument specifying where the target address is located (*e.g.,* `jmp rax` jumps to the location specified by `rax`) instead of the argument being the target address as in direct jumps (*e.g.,* `jmp 0xdeadbeef` jumps to the address `0xdeadbeef`). To measure the cost of indirect jumps compared to direct jumps, we call functions indirectly by function pointers as shown in Figure 2b.

Security mitigations for speculation vulnerabilities (*e.g.,* Spectre [54]) significantly increase the cost of indirect jumps and make it expensive to call a chain of BPF extensions via a loop of indirect jumps. This is because indirect jumps are typically optimized by speculation—modern CPU architecture speculates the target address for indirect jumps in the pipeline for optimal performance; however, with Spectre [54] and Meltdown [59] which exploit hardware design flaws of speculation (to leak information from unintended speculative paths via micro-architectural side channels), the speculation is disabled. Fixing these vulnerabilities without performance impact remains as an open problem.

We benchmark the *de facto* software mitigation for speculation vulnerabilities—Retpoline [21], and discuss the other alternative mitigations proposed by recent research as they are not readily available for benchmarking.

Retpoline. Listing 1 shows the implementation of retpoline. The key idea of retpoline is to prevent CPUs from speculating the target address of an indirect jump. Retpoline works by placing the target address on top of the stack and calls a *thunk* to jump to target by a return instruction (line 7). The loop in the middle (line 2–4) is never executed, and the purpose of it is to fill the instruction pipeline with dummy values to prevent the CPU from speculating the actual target address, effectively nullifying the substantial performance benefits of speculation along with the risks.

As shown in Table 1, without retpoline, the indirect jump with a function pointer takes 2.6 cycles which is as fast as direct calls in Figure 2c. The is due to the high accuracy (up to 98% [76]) of branch target predictors in modern CPUs. However, with retpoline enabled, an indirect jump takes 32

```
    call retpoline_call_target
2:
    lfence /* stop speculation */
    jmp 2b
retpoline_call_target:
    lea 8(%rsp), %rsp
    ret
```

Listing 1. Retpoline assembly

extra cycles making it 13.3× slower because retpoline forces a branch prediction miss on every indirect jump and stops the CPU from speculatively executing more instructions. Also, retpoline itself introduces more instructions including jumping to the retpoline thunk, moving the target function on the stack and returning to the target function. In particular, jumping to the thunk takes 14.79% of the sampled cycles, moving the target function on the stack takes 8.87% and returning to the target function takes 76.33%.

Alternative mitigations. Recent work [15, 31, 55] proposes to use indirect branch promotion as an optimization to avoid indirect jumps, and to use control-flow integrity (CFI) to regulate speculation targets. Indirect branch promotion needs to predict the indirect branch target and convert an indirect jump into a comparison and a direct jump. Its performance depends on the prediction. An indirect jump with retpoline is still taken for misprediction.

CFI-based approaches need hardware supports to speculate only known targets for protection against Spectre-like attacks. When misspeculation happens, an expensive serializing instruction (*e.g.,* `lfence`) is used.

Therefore, even with those techniques, indirect jumps will still be much expensive than direct jumps.

3.2.2 Loop cost.

A loop for executing a BPF extension chain includes the following operations. An index counter represents the index in the current iteration. The counter needs to be incremented by 1 to find the next program at the end of each iteration. Within each iteration, the index counter is used to retrieve the corresponding element in the array. Components of a loop result in more instructions and branches compared to the unrolled loop. To measure the cost of the loop, we manually unroll the loop in Figure 2a and call functions via function pointers as shown in Figure 2b. We compare the two cases to obtain the extra cycles per iteration that the loop contributes which are 12.4 (26.4% of the extension call) with retpoline and 6.7 (72% of the extension call) cycles without. Therefore, the loop can cause overheads for executing BPF extension chains. Note that retpoline stalls CPU pipelines and magnifies the cost.

We further investigate and break down the cost of a loop. The cost of a loop per iteration includes four parts, (1) checking if the loop condition holds to determine if the loop is over (usually on the index is less than the array size), (2) incrementing the index counter, (3) loading the extension pointer from the array based on the index counter, and (4) jumping to the beginning of the loop. We use `perf` to profile the loop by CPU sampling. Checking the loop condition takes 28.91% of the sampled cycles. 51.76% of the cycles are spent incrementing the index counter stored on the stack. Loading extensions from the array takes 10.99% of the sampled cycles. Finally, 8.34% of the cycles are spent jumping back to the beginning of the loop. The common approach to mitigate the cost is to unroll the loop. However, this obvious solution of loop unrolling can not be done, if the execution of BPF extensions are tightly coupled to the verification and admission of them.

3.2.3 Memory access cost.

To use tail calls, a program needs to firstly look up the target program from a program array and jump to the target. Looking up the target program is essentially the same operation as one of the loop operations – loading the extension from the array based on the index counter (Section 3.2.2). These two operations both read the target program address from an array and therefore, need memory access. According to our measurement of loops, we find that the memory access cost can take up 11% of the loop cost which is 1.36 cycles with retpoline.

3.3 Discussions and Implications

The chain overhead can be significant when the BPF extensions are placed at a performance-critical path. For instance, 54% performance regression is reported for XDP packet filtering when there are only two indirect function calls (jumps) per packet [29]. When the chain is long, the overhead will be accumulated and impact application performance. Some real-world use cases have much longer BPF extension chains than the three-extension chain used in our measurement. For example, `systemd` [24] installs tens of seccomp BPF filters for different system services, which has significant performance overhead (as shown in Section 5.1.1).

The chain overheads come from the fact that BPF extensions are individually verified and loaded into the kernel. We argue that the execution of these extensions should not be constrained by how they are verified and loaded. Instead, the execution of the chained BPF programs can be *collectively* optimized in the kernel. Specifically, we can see that replacing indirect jumps with direct jumps, unrolling loops, and saving memory accesses, and each leads to in remarkable performance improvements (Table 1). Therefore, in KFuse, we start from supporting these three optimizations.

4 KFuse Design and Implementation

4.1 Overview

KFuse is a in-kernel framework for *collectively* optimizing verified BPF extensions that are installed at the same hooking point, in the pattern of chains. In essence, it separates the concerns of verification and execution of BPF extensions.

BPF extensions that are installed by different principals or at different points of time are verified *independently* to ensure safety. KFuse is invoked after the verification to optimize the execution performance.

Currently, KFuse supports three generic optimizations: (1) converting indirect jumps into direct jumps, (2) unrolling loops, and (3) saving memory accesses by rewriting BPF tail calls. It can be easily extended to incorporate other optimizations, such as domain-specific ones.

A BPF extension is loaded from user space, verified by the kernel, and optionally just-in-time (JIT) compiled to native machine code. Then, the target kernel subsystems (*e.g.*, seccomp, tracepoint, and cgroup) can install it by attaching it to the corresponding hook points. KFuse can be invoked dynamically with the input of a chain of BPF extensions; it generates one optimized BPF extension that "fuses" the original BPF extension chain and replaces it with the fused extension at the hook point. Currently, we invoke KFuse after a new BPF extension is loaded and verified. KFuse fuses BPF programs at the BPF-instruction level, rather than native machine code—the fused BPF extension can still be (optionally) JITed.

Listing 2 shows how KFuse is invoked in seccomp. It invokes the currently-attached BPF filter (`current_filter`) with the newly-admitted BPF filter (`prepared`). The current filter can be a fused BPF program.

```
prepared = seccomp_prepare_user_filter(filter);
if (current_filter != NULL)
    merge_bpf_progs(current_filter->prog,
        prepared->prog, SECCOMP_RETURN_POLICY);
```

Listing 2. KFuse for BPF seccomp system call filters

Figure 3 further illustrates the process, where Figure 3a illustrates the original BPF chains at different hook points and Figure 3b illustrates the fused BPF extension when KFuse is used. KFuse first fused `b`, `c`, and `d` at the hook point `H2` into an optimized program, `bcd`, and `f` and `g` at `H3` into `fg`. When a new BPF extension `E` is loaded and hooked at `H3`, it fuses `fg` and `E` into `fgE`.

The fused program maintains the safety properties enforced by the BPF verifier, which can be proved by induction-based on the safety properties of the original BPF program (ensured by the verifier). On the other hand, the fused program is not limited by the size or complexity constraints required by the BPF verifier to bound the verification time.

KFuse is an in-kernel mechanism that are transparent to user space. It preserves the original extension structure and creates the illusion of a chain of multiple BPF extensions. BPF extensions are loaded and installed in exactly the same way and no user-space change is needed.

Why not fuse at user space? Fusing the BPF extensions at user space is not an option. The fused BPF extension could

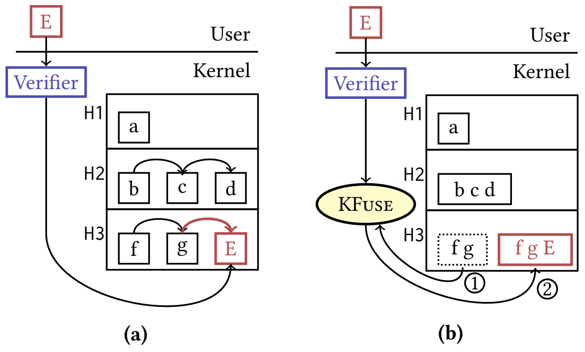


Figure 3. Illustration of KFuse: (a) the original BPF extensions; (b) the optimized BPF extensions with KFuse.

be difficult for the existing BPF verifier to verify due to the increased complexity. Furthermore, certain BPF use cases (*e.g.*, seccomp) do not allow dynamic updating installed BPF extensions for security reasons.

4.2 Optimizing a loop of BPF extensions

As discussed in Section 3, using a loop to iteratively execute loaded BPF extensions is the most common implementation pattern to support BPF extension chains. KFuse optimizes such patterns by merging the BPF extension chain into one large BPF extension by unrolling the loop and converting indirect jumps into direct jumps.

Conceptually, such an optimization is as simple as concatenating two BPF extensions. Two important issues need to be taken care of: *return instructions* and *jump offsets*.

4.2.1 Rewriting return instructions into direct jumps. A return instruction terminates the execution of a BPF program. To merge two BPF programs `A` and `B` into `A+B`, KFuse needs to automatically rewrite the return instructions in `A` into two conceptual instructions, one "jump" instruction destined to the first instruction in `B` and one "update" instruction for updating an aggregated return value. The "jump" instruction can be implemented using a *direct* jump. The "update" instruction may need multiple instructions to implement.

KFuse locates every return instruction in the BPF program and replaces it with a direct jump instruction destined to the start of the next BPF program. If the BPF program is the last one in the chain, KFuse replaces the return instructions with direct jumps to a global return instruction (which is the last instruction of the merged program, created by KFuse).

Figure 4 shows an example of merging two BPF programs for seccomp, `FILTER_1` and `FILTER_2`. KFuse replaces each return instruction in `FILTER_1` with a direct jump instruction destined to `FILTER2_START`, and replaces each return instruction in `FILTER_2` with a direct jump destined to `END`.

4.2.2 Updating return values. Updating return values needs to follow the loop semantics to ensure the fused program has the same semantics as the original loop. Different applications aggregate the return value of individual BPF programs in the loop differently, as shown in Figure 1. We refer to the aggregation semantics as a *return value policy.*

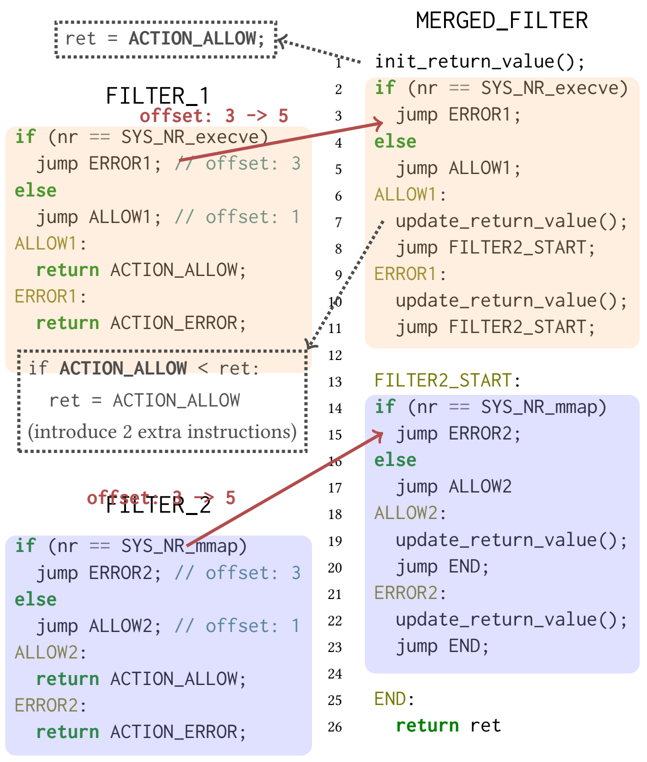
Currently, KFuse supports all the loop semantics of BPF use cases in Linux, including returning the minimal value (seccomp, Figure 1a), returning the bitwise-AND of the values (tracepoint and cgroup, Figures 1b and 1c), and returning the first non-zero values (`tc` and `lsm-bpf`). We discuss the return value policies of seccomp, tracepoint, and cgroup:

- **Seccomp.** A seccomp filter returns an action such as allowing the system call or killing the process. The more restrictive the action is, the smaller the return value is. Consequently, we implemented a return value policy that initializes the return value to the maximum integer value and only updates the new return value only if it is smaller than the current value to match the semantics of seccomp (Figure 1a).
- **Tracepoint.** The final return value of tracepoint BPF programs is calculated by performing an AND operation on each return value of the individual programs. The return value policy is implemented by initializing the return value to 1 and updating it via an AND operation on the current and new values.
- **Cgroup.** When a network packet arrives to a process in a cgroup, programs attached to the cgroup will be executed to determine the action for the packet based on the return values of BPF programs. The final value is also calculated by an AND operation like tracepoint. Therefore, the return value policy is the same as the one for tracepoint (initializing the return value to 1 and updating it via an AND operation on the current and new values).

KFuse initializes the global return value stored in temporal storage, and we use `BPF_REG_AX` in our implementation. It updates the return value at the locations of the original return instructions based on the return value policy (which is an input of KFuse). The global return value will be returned by the fused BPF program. In Figure 4, the return values are updated based on seccomp's return value policy that returns the smallest value (the most restrictive policy).

4.2.3 Recalculating jump offsets.

Jump offsets need to be recalculated based on return instructions because replacing one single return instruction with multiple instructions changes the extension size and misaligns the original jump offsets. KFuse calculates the new offset by counting the number of return instructions between a jump and its destination. For instance, at line 3 in Figure 4, the jump offset is updated to 5 from 3 because there is a return instruction between the jump and its destination in the original program, assuming that updating the return value takes 2 additional instructions.



```
ret = ACTION_ALLOW;
```

FILTER_1

```
if (nr == SYS_NR_execve)
  jump ERROR1; // offset: 3
else
  jump ALLOW1; // offset: 1
ALLOW1:
  return ACTION_ALLOW;
ERROR1:
  return ACTION_ERROR;
```

```
if ACTION_ALLOW < ret:
  ret = ACTION_ALLOW
```

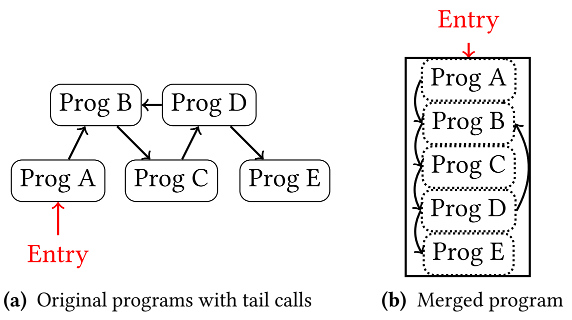
FILTER_2

```
if (nr == SYS_NR_mmap)
  jump ERROR2; // offset: 3
else
  jump ALLOW2; // offset: 1
ALLOW2:
  return ACTION_ALLOW;
ERROR2:
  return ACTION_ERROR;
```

MERGED_FILTER

```
1   init_return_value();
2   if (nr == SYS_NR_execve)
3     jump ERROR1;
4   else
5     jump ALLOW1;
6   ALLOW1:
7     update_return_value();
8     jump FILTER2_START;
9   ERROR1:
10    update_return_value();
11    jump FILTER2_START;
12
13  FILTER2_START:
14  if (nr == SYS_NR_mmap)
15    jump ERROR2;
16  else
17    jump ALLOW2
18  ALLOW2:
19    update_return_value();
20    jump END;
21  ERROR2:
22    update_return_value();
23    jump END;
24
25  END:
26    return ret
```

Figure 4. KFuse merges two seccomp BPF filters by rewriting the return instructions and updating jump offsets. Dotted boxes: the actual instructions of the conceptual instruction.



(a) Original programs with tail calls (b) Merged program

Figure 5. KFuse rewrites tail calls into direct jumps to merge BPF programs that are connected by tail calls.

4.3 Optimizing tail calls

A BPF program can tail-call other BPF programs to form a chain (*e.g.,* Prog A, B, C, D and E in Figure 5a). To do so, the caller first reads the address of callee from a program array and then performs an indirect jump to the target address. Given the high cost of indirect jumps with retpoline, the JIT compiler, if enabled, will try to optimize the tail call by rewriting the indirect jumps into direct jumps if possible. Note that it is not always possible, *e.g.,* on x86, the target address of the direct jump instruction is stored in a 32-bit signed integer so jumping further than a 32-bit signed integer

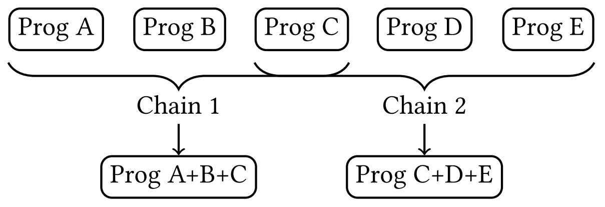


Figure 6. Extension is decoupled from the execution.

can only be indirect. Also, this requires to enable JIT, which is not always available or secure [65].

BPF programs can use tail calls to form a bounded call loop by tail-calling back to the caller (*e.g.,* Prog B, C and D in Figure 5a). BPF verifier imposes a constraint on the number of tail calls (`MAX_TAIL_CALL_CNT`) to ensure that the program does not execute infinitely by a call loop.

KFuse eliminates the need of a memory access that reads the jump target from the extension array by merging the target extension into the caller. Moreover, KFuse converts the indirect jumps to direct jumps if any. Note that this optimization is done independently from JIT, and is before JIT if it is enabled. Therefore, it is available to architectures without a JIT compiler implementation.

KFuse performs a depth-first search to figure out all target programs when a program that uses tail calls is loaded. KFuse then merges them into one single program and converts the original tail calls to direct jumps to the offset at which the target program is in the merged program, as shown in Figure 5b. A backward jump is permitted in the merged program. The merged program still always terminates and is a directed cyclic graph (Section 4.5.1), because KFuse enforces the constraint of tail-call count (`MAX_TAIL_CALL_CNT`)—calling repeated programs can be seen as an edge to a new node (*i.e.,* `MAX_TAIL_CALL_CNT` nodes are allowed at maximum). So, it is equivalent to unrolling the bounded loops.

4.4 Transparency to user space

KFuse is fully transparent to user space, by maintaining the original BPF extension structures. It decouples the execution from other operations and only adds a pointer to the fused extensions, as illustrated in Figure 6. Therefore, user space APIs, such as looking up, dumping or removing an extension, are not affected. From the user space view, these BPF extensions are still maintained as a chain but only the fused extension is executed when the chain is invoked. On the other hand, the fused extension needs to be regenerated whenever the chain is updated.

KFuse also supports sharing of BPF extensions, when a BPF extension needs to be attached to different chains (*e.g.,* Prog C in Figure 6). For example, seccomp BPF filters can be shared by different processes. KFuse merges the shared extension to generate different fused extensions. In our experience, BPF extensions are small in size and the additional memory usage is not concerned, as measured in Section 5.6.

Property	**Purpose**	**Maintained**
Directed acyclic graph (DAG)	Safety	✓
No unreachable code	Safety	✓
No invalid memory access	Safety	✓
No invalid jump	Safety	✓
Constraint		
`MAX_BPF_STACK`	Size	✓
`MAX_CALL_FRAMES`	Size	✓
`MAX_TAIL_CALL_CNT`	Complexity	✓
`BPF_COMPLEXITY_LIMIT_JMP_SEQ`	Complexity	✗
`BPF_COMPLEXITY_LIMIT_INSNS`	Complexity	✗
`MAX_USED_MAPS`	Size	✗
`BPF_MAX_SUBPROGS`	Complexity	✗

Table 2. Properties and constraints imposed the verifier. KFuse preserves all the safety properties.

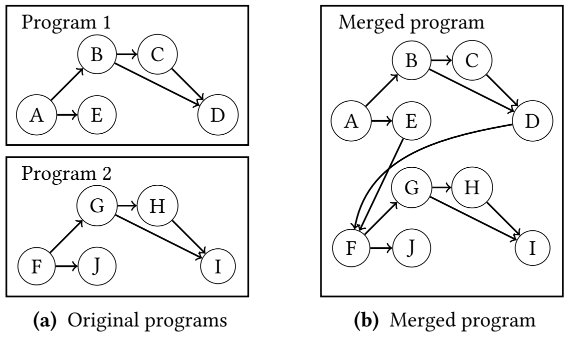


Figure 7. KFuse merges two BPF programs (Programs 1 and 2) that have been already verified to be a DAG by connecting the exits of Program 1 (vertices with 0 outdegree, *i.e.,* D and E) to the entry of Program 2 (the vertex with 0 indegree, *i.e.,* F). The merged program is still a DAG.

4.5 Maintaining Safety Properties

Since kernel extension programs are provided from the less privileged user space, the verifier imposes various constraints on them ((*f*) Section 2.1). The BPF verifier ensures the safety properties listed in Table 2 hold, including that programs are directed acyclic graphs (DAG), that programs have no loops/unreachable code and that programs have no invalid memory accesses or jumps. These properties ensure the safety of the kernel when loading user-supplied BPF programs. The verifier imposes a number of constraints to satisfy these properties, also shown in Table 2.

4.5.1 Satisfying safety properties.

The fused BPF program maintains all safety properties enforced by the verifier on every original BPF program. This can be proven by induction. We sketch the ideas below.

- *Directed acyclic graph (DAG).* Instructions of a BPF program can only be a DAG (no backward jumps or unbounded loops). The DAG property guarantees the termination of the program and also ease the detection of unreachable code. KFuse merges two programs by connecting the exits of one program to the start of the other. The fused BPF program is proven to be a DAG—a graph is a DAG iff it can be topologically ordered; the fused program can be topologically ordered by appending the topological order of the second program to the topological order of the first program.

- *No unreachable code* A BPF program cannot contain unreachable code—it needs to be a connected graph. By connecting the exits of the first program (vertices D and E) to the start of the second program (vertex F), the merged program (Figure 7b) is still connected and contains no unreachable code: every vertex in program 1 is reachable from A, every vertex in program 2 is reachable from F, and F is reachable from D or E.
- *No invalid memory access or malformed jumps* A BPF program cannot access invalid memory region (*e.g.,* memory beyond the stack) or include malformed jumps (*e.g.,* backward or out-of-bound jumps). Every program is individually verified before being merged by KFuse. KFuse does not change its memory access and only adds jumps to well-specified destination.

4.5.2 Ignoring constraints for bounding verification time.

KFuse is not limited to the constraints imposed by the BPF verifier, if the constraints are solely purposed for bounding the verification time. This is because each individual BPF program is already verified and the fused BPF program does not need to be verified again. The safety properties are maintained by construction.

Table 2 shows those constraints that are used by the verifier. Specifically, BPF_COMPLEXITY_LIMIT_JMP_SEQ and BPF_COMPLEXITY_LIMIT_INSNS are used to bound the program size and complexity; MAX_USED_MAPS is used to limit the number of maps, and BPF_MAX_SUBPROGS is used to limit the subprograms in a BPF program.

4.5.3 Domain-specific constraints.

Some kernel subsystems also impose domain-specific constraints. For example, cgroups and tracepoints both limit the number of BPF programs that can be attached to the same hook point by BPF_CGROUP_MAX_PROGS and BPF_TRACE_MAX_PROGS. These constraints are agnostic to KFuse and KFuse always respect them. This is because the constraints are placed on the original program structure (Section 4.4). If the constraints are not satisfied, KFuse will not be invoked. We note that these constraints are not for safety but for performance purposes as they bound the total execution time.

4.6 Implementation

KFuse is implemented in around 500 lines of C code, on top of the Linux kernel (v5.9). As discussed in Section 4.1, the main interface is merge_bpf_progs(), which takes three input parameters, the pointers of the original BPF programs, and a policy flag that specifies the return value policy, and returns a pointer of the fused program, as shown below.

```
struct bpf_prog* merge_bpf_progs(
    struct bpf_prog* old_prog,
    struct bpf_prog* new_prog,
    enum return_policy);
```

KFuse uses BPF_REG_AX (a special register reserved for the kernel) to hold the temporal return value, which is updated according to the return value policy. The merged program has two-instruction prologue that initializes the return value. Each return instruction is replaced with 5 instructions for the Seccomp return policy and 3 instructions for the tracepoint and cgroup return policy. Finally, the merged program is appended with a two-instruction epilogue which loads the return value from BPF_REG_AX to BPF_REG_A and returns.

Integrating KFuse. Kernel subsystems can choose to integrate KFuse. The integration is straightforward with a few lines of code. We integrate KFuse in three kernel subsystems, including seccomp, tracepoint, and cgroup. We apply KFuse to seccomp with 20 lines of code (LoC), tracepoint with 28 LoC and cgroup with 34 LoC. We implement a sysctl configuration to allow users to enable/disable KFuse.

5 Evaluation

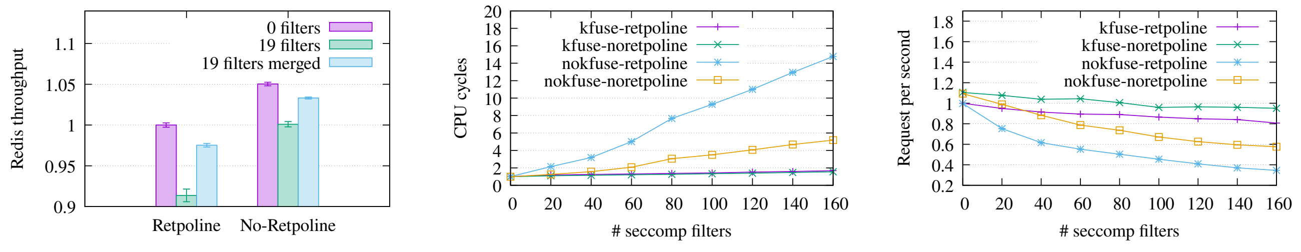
We measure the effectiveness of KFuse in optimizing performance, as well as its overhead in terms of the time it takes for optimization and additional memory cost. To measure performance, we run performance benchmarks using the three Linux subsystems that integrates KFuse (seccomp, tracepoint and cgroup). All experiments are run on a single server with a 16 core Intel Xeon Silver 4110 CPU at 2.10GHz and 64 GB of memory. We run the experiments in KVM virtual machines. Each VM is configured with 2 VCPUs and 8 GB memory, and the kernel is compiled with the microVM configuration. We use wrk [14] and memtier_benchmark [17] as benchmark clients for NGINX and Redis. We also evaluate the cases where the kernels are compiled without retpoline, which gives a lower bound of the KFuse effectiveness.

Overall, KFuse effectively improves the performance for real-world workloads (7% throughput increase for Redis deployed by systemd). The performance benefits of KFuse grows linearly along with the length of chains.

5.1 Seccomp

We measure the effectiveness of KFuse in improving the system call performance under seccomp BPF filters. We conduct three experiments:

- We use KFuse to optimize the seccomp BPF filters launched by systemd which provides application sandboxing using seccomp. We run Redis and NGINX as real-world applications to assess the benefit of using KFuse on *existing* applications and BPF use cases.
- We conduct a scaling analysis on both system call performance and application performance, with increasing numbers of seccomp BPF filters



(a) Redis throughput (b) Latency of `getpid` (c) NGINX throughput

Figure 8. (a) The throughput of Redis with 19 seccomp filters installed by `systemd`. **(b)(c)** The latency of `getpid` and NGINX with different numbers seccomp filters installed for scaling analysis. The results are normalized to baseline, 0 seccomp filters and retpoline enabled.

- We present the benefit of KFuse for *incremental tightening* which installs seccomp filters during application execution to tighten the system call policy. Incremental tightening is meant to be one of the new BPF use cases that lead to a long chain of BPF extensions and motivate the need to reduce the cost.

5.1.1 Systemd BPF filters.

Systemd provides a sandboxing feature that restricts the system calls the sandboxed applications can invoke. We benchmark Redis launched by `systemd`. Redis is shipped on Ubuntu with a `systemd` configuration that installs 19 seccomp BPF filters sandboxing the service including loading kernel modules, gaining new privileges and modifying control groups. Systemd also installs filters for supporting 32-bit systems (*i.e.,* i386). These filters are sized from 6 instructions to 58 instructions with an average 19.6 instructions. Figure 8a shows the throughput of Redis, with the original 19 filters and with the fused filter by KFuse. The results are normalized by the Redis throughput with no BPF filter. We can see that the 19 filters installed by `systemd` lead to non-negligible overhead (≈ 10% throughput decrease). KFuse can significantly reduce the overhead (with only 3%) by speeding up the end-to-end BPF execution time from 957 to 148 nanoseconds (reducing 85%) with the same security policy of original filters.

5.1.2 Scaling analysis.

We conduct a scaling analysis to measure the performance overhead of different number of seccomp filters and the benefit of using KFuse. We create a simple BPF filter with 4 instructions: checking if the system call number equals 450, if true, kill the process. Otherwise, allow the call. The system call numbered 450 does not exist, so all the system calls will be allowed.

Figure 8b shows the execution time of `getpid` with different numbers of BPF filters. With 20 filters, KFuse can speed up `getpid` performance by 1.8× and 1.2× with and without retpoline. With 160 filters, KFuse can speed up `getpid` performance by 8.8× and 4.9× with and without retpoline.

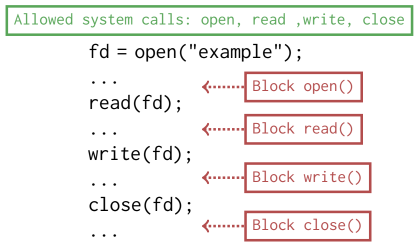

```
Allowed system calls: open, read ,write, close
fd = open("example");
...                     Block open()
read(fd);
...                     Block read()
write(fd);
...                     Block write()
close(fd);
...                     Block close()
```

Figure 9. Incremental tightening blocks system calls during program execution. This simplified program only uses 4 system calls once, `open`, `read`, `write`, and `close`. The system call is blocked when it will not be again.

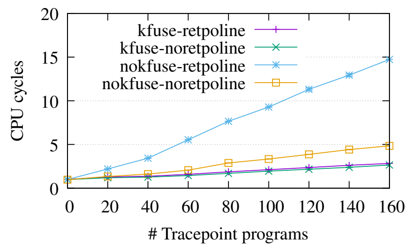
We also compare the two cases with retpoline disabled (kfuse-noretpoline and nomerger-retpoline). We find that the execution without the KFuse has 4.7× more instructions, 2.6× more branches and 7.8× more branch misses. The additional instructions and branches come from the loops and branch-misses come from retpoline.

Figure 8c shows the throughput of NGINX with different numbers of seccomp filters installed. The throughput drops linearly in both cases with and without retpoline. With 20 filters, KFuse increases throughput by 1.3× with retpoline and 1.1× without. With 160 filters, KFuse increases throughput by 2.3× with retpoline and 1.6× without.

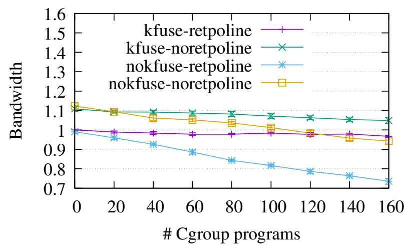
5.1.3 Incremental tightening.

Incremental system call tightening is a fine-grained temporal system call specialization [46], illustrated in Figure 9. It installs multiple BPF filters based on the application's execution phases.

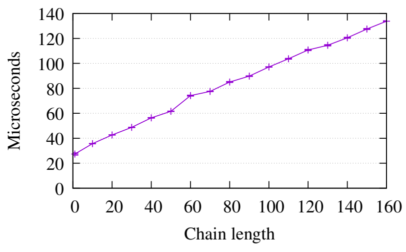
We generate the seccomp BPF filters for different phases of NGINX and Memcached by dynamically tracing their runtime behavior continuously. There are non-deterministic system calls such as those related to timer or signals. We always allow non-deterministic system calls and only incrementally tighten deterministic system calls. We modify the C library to load the profile and install a seccomp filter when a system call will no longer be needed.



(a) Latency of `getpid`

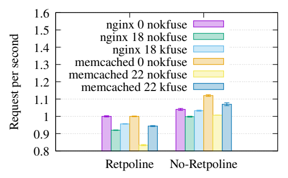


(b) Network bandwidth

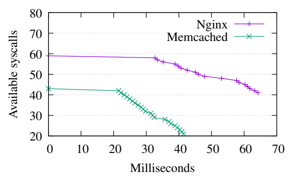


(c) Merging latency

Figure 10. Performance measurement of **(a)** `getpid` with different numbers of Tracepoint programs and **(b)** IPerf with different numbers of Cgroup firewall rules. The results are normalized to the baseline, 0 programs and retpoline enabled. **(c)** shows the time KFuse needs to update extensions on a chain.



(a) Throughput of NGINX and Memcached



(b) Number of allowed system call during execution

Figure 11. (a) Throughput of NGINX and Memcached with 18 and 22 seccomp filters by incremental tightening. The results are normalized to the baseline, 0 seccomp filters and retpoline enabled. **(b)** The number of allowed system calls for NGINX and Memcached over time as incremental tightening blocks system calls during execution.

We install 18 seccomp filters for NGINX and and 22 for Memcached. Figure 11b shows the number of allowed system calls over time for each application. Figure 11a shows that the application performance drops 7%–16% due to the chain of filters. By merging these filters, KFuse eliminates the overheads of calling extensions by 45%–66%.

5.2 Tracepoint

Tracepoint enables live kernel debugging and profiling. Multiple BPF programs can be attached to a single tracepoint and all of them are executed when an event is triggered. To experiment with realistic workloads, we use a tracepoint BPF program that not only performs arithmetic instructions but also write data to a BPF map. The program counts the number of times that a system call, `getpid`, is called and write the counter to a map. Figure 10a shows the latency of `getpid` with various numbers of programs attached. With 20 programs, KFuse can speed up latency by 1.71× with retpoline and 1.12× without. With 160 programs, KFuse can speed up latency by 5.19× with retpoline and 1.82× without. In the average case of 80 programs, KFuse can speed up latency by 4.09× with retpoline and 1.7× without.

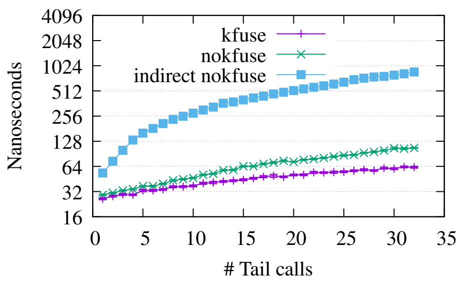


Figure 12. Time to call different numbers of tail calls. Tail calls are optimized to direct jumps unless specified.

5.3 Cgroup

Cgroup provides the mechanism for users to attach multiple BPF programs to implement firewall rules. We attach multiple BPF programs with type `BPF_CGROUP_INET_INGRESS` to experiment with the scenario where multiple firewall rules are applied. The BPF programs simply permit the packet for performance evaluation. We use iPerf3 [9] to measure the TCP bandwidth with various numbers of programs attached. When attaching less than 60 programs, we do not observe obvious performance enhancements. With 60 programs, KFuse improves the bandwidth by 10% with retpoline and 3% without. With 160 programs, KFuse improves the bandwidth by 31% with retpoline and 11% without.

5.4 Tail call

To measure KFuse's saving of tail call rewriting, we use the benchmark used by Joly *et al.* [49] for evaluating tail call overhead, in which a BPF program tail-calls various number of other BPF programs and forms chains with different lengths, from 1 to 32. We verify that these tail calls are optimized to direct calls and thus are not slowed down by retpoline. We also evaluate the case when tail calls are not optimized and are indirect jumps. Figure 12 shows the average of 50 iterations in three cases. When tail calls are optimized to direct jumps, KFuse saves memory access cost. The average speed-up (16 tail calls) is 1.4× with the minimum (1 tail call) and maximum (32 tail calls) being 1.1× and 1.7×. When tail calls are indirect jumps, KFuse saves both memory access and indirect jump cost. The average speed-up (16 tail calls) is 16.1× with the minimum (1 tail call) being 2× and maximum (32 tail calls) being 32.88×.

5.5 Cost of updating a chain

We measure the time KFuse takes to fuse a BPF program chain into an optimized BPF program. KFuse works dynamically by fusing the existing program (the merged chain) with the new program. Specifically, KFuse needs to combine the new program with existing one, reallocate memory for the merged program and generate JITed image for the merged program so the time to merge depends on the length of chain (the size of existing program) and the new program. We measure the time by loading up to 160 programs. As shown in Figure 10c, the latency starts from ≈ 27 microseconds when there are only 1 program on the chain and grows to ≈ 138 microseconds when there are 160. When there are 80 programs on the chain, the merging time is ≈ 85 microseconds.

5.6 Memory usage

KFuse needs to keep the original BPF bytecode to support sharing of BPF programs and maintain transparency to the user space (Section 4.4). This could lead to additional memory overhead. To quantify the overhead, we measure the size of 366 real-world BPF programs from six projects collected by [45] (including ovs, linux, prototype-kernel, suricata, bpf_cilium_test and cilium). The average BPF extension size is merely 4.3 KB, with the median and the 90-th percentile being 0.9 KB and 13.8 KB, respectively. Given the abundant memory of modern computers, the additional memory usage is acceptable.

6 Related work

BPF extensions have achieved phenomenal adoptions in recent years, including security [33, 37, 41, 42] , storage [27, 56, 67, 73], profiling [2, 12, 39] and networking [10, 28, 34, 35, 40, 48, 60, 66, 68, 71, 74, 75]. Recent advances on safe BPF infrastructures and implementations [30, 44, 45, 52, 53, 58, 62, 63, 72, 75] further eliminate the barrier of adoptions. KFuse respects the safety properties ensured by BPF verifiers, while optimizing the performance of BPF extension chains.

A main optimization of KFuse is to convert indirect jumps into direct jumps. The costs of indirect jumps come from the mitigation of Spectre [54], a recent class of hardware vulnerabilities, which degrades the efficiency of conditional branches. In addition to retpoline [21], recent research proposals for Spectre mitigation includes control-flow hardening [43, 55], page table separation [36] or cache isolation [51] to remove the side channels. Recently, Intel released indirect branch restricted speculation (IBRS) [23] and enhanced IBRS [23], new hardware features to prevent less-privileged processes from controlling the branch target predictor in a more-privileged mode. However, these features either incur huge performance overheads or are only available on new CPUs. We expect the overhead of indirect jumps to remain high in the near future, because free speculation is no longer an option [38, 64].

A few efforts have been made to reduce the overhead of indirect jumps. JumpSwitches [31] optimizes retpoline by branch promotions, which depends on an effective target predictor. A few Linux patches [1, 25] try to convert unnecessary indirect jumps to direct jumps to improve XDP performance [29]. The BPF JIT compiler optimizes tail calls to direct jumps [18] if the target address is within an offset that can be expressed as a 32-bit integer. BPF trampoline [4, 8] is a mechanism to directly call BPF programs from kernel by a dynamically generated code image which needs to be regenerated when an extension program is added or removed. Optimizations such as trampoline [8] or converting tail calls to direct jumps [18] require BPF programs to be JIT-compiled, which may not be available. Differently, KFuse focuses on eliminating the indirect jumps across the chained BPF extensions. As we shown in the paper, the overhead of chained extension can be significant as the BPF programs are typically small in size. KFuse does not rely on JIT compilation.

7 Conclusion and Discussion

We demonstrate that multiple BPF programs in a chain can be collectively optimized after each of them is verified. Such optimization can lead to remarkable performance gains. Specifically, when the BPF programs are small in size and the chain is long, the overhead of indirect jumps and loops can be as high as 85% of the end-to-end execution time, leading a 7% application performance loss. We have already seen long chains of BPF extensions in real deployments (*e.g.,* 19 Seccomp BPF filters deployed by `systemd`) and we expect such long chains to be even more common in future use cases, given the explosion of BPF use cases in recent years and the need of fine-grained, domain-specific policies.

We believe that more advanced cross-optimizations for verified BPF programs (such as application-specific optimizations) can be implemented, beyond the basic optimizations currently supported by KFuse. For example, redundant code (*e.g.,* checks [77]) across the BPF extension programs can be removed, in the scope of the entire BPF chain, to further improve the performance; BPF instructions can be reordered across the BPF extension chain (*e.g.,* by placing hot code early in the fused BPF program to save subsequent computation). Those optimizations can be supported in the current KFuse framework. Further, multiple BPF programs at different hooking points are often orchestrated to collectively implement a feature (*e.g.,* socket filters hooked at cgroup and traffic control [5]). Those BPF programs can potentially be optimized together, while maintaining the safety properties.

References

[1] Add static_call(). https://lwn.net/Articles/819311/. Accessed: 09/04/2021.
[2] BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc. Accessed: 09/04/2021.
[3] Beautifying syscall args in 'perf trace'. http://vger.kernel.org/~acme/perf/linuxdev-br-2018-perf-trace-eBPF/. Accessed: 09/04/2021.
[4] The bpf dispatcher. https://lore.kernel.org/bpf/20191211123017.13212-1-bjorn.topel@gmail.com/. Accessed: 09/04/2021.
[5] Bpf map tracing: Hot updates of stateful programs. https://linuxplumbersconf.org/event/11/contributions/942/. Accessed: 09/04/2021.
[6] cgroups(7) — linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html. Accessed: 09/04/2021.
[7] Evaluation of tail call costs in ebpf. https://www.linuxplumbersconf.org/event/7/contributions/676/attachments/512/1000/paper.pdf. Accessed: 09/04/2021.
[8] Introduce bpf trampoline. https://lwn.net/Articles/804937/. Accessed: 09/04/2021.
[9] iPerf - The ultimate speed test tool for TCP, udp and sctp. https://iperf.fr/iperf-doc.php. Accessed: 09/04/2021.
[10] Linux Native, api-aware networking and security for containers. https://cilium.io/. Accessed: 09/04/2021.
[11] Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt. Accessed: 09/04/2021.
[12] Linux system exploration and troubleshooting tool with first class support for containers. https://github.com/draios/sysdig. Accessed: 09/04/2021.
[13] Lsm bpf programs. https://www.kernel.org/doc/html/latest/bpf/bpf_lsm.html. Accessed: 09/04/2021.
[14] Modern http benchmarking tool. https://github.com/wg/wrk. Accessed: 09/04/2021.
[15] net: mitigate retpoline overhead. https://lwn.net/Articles/773985/. Accessed: 09/04/2021.
[16] new seccomp mode aims to improve performance. http://kernsec.org/pipermail/linux-security-module-archive/2020-June/020706.html. Accessed: 09/04/2021.
[17] Nosql redis and memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark. Accessed: 09/04/2021.
[18] Optimize bpf tail calls for direct jumps. https://lore.kernel.org/bpf/CAEf4BzYPLVHpc=EifKZP7wcfeWpzbENsD9MOb_UN=_48wpW24Q@mail.gmail.com/T/. Accessed: 09/04/2021.
[19] [PATCH net-next 8/9] net: filter: rework/optimize internal BPF interpreter's instruction set. https://lore.kernel.org/netdev/1395404418-25376-9-git-send-email-dborkman@redhat.com/. Accessed: 09/04/2021.
[20] Performance Benchmark Analysis of Egress Filtering on Linux. https://kinvolk.io/blog/2020/09/performance-benchmark-analysis-of-egress-filtering-on-linux/. Accessed: 09/04/2021.
[21] Retpoline: A branch target injection mitigation. https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf. Accessed: 09/04/2021.
[22] Secure computing with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt. Accessed: 09/04/2021.
[23] Speculative execution side channel mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf. Accessed: 09/04/2021.
[24] systemd. https://www.freedesktop.org/wiki/Software/systemd/. Accessed: 09/04/2021.
[25] Testing Hellwig "dma-direct-calls" patchset. https://github.com/xdp-project/xdp-project/blob/master/areas/dma/dma01_test_hellwig_direct_dma.org. Accessed: 09/04/2021.
[26] Using the linux kernel tracepoints. https://www.kernel.org/doc/html/latest/trace/tracepoints.html. Accessed: 09/04/2021.
[27] When ebpf meets fuse. https://events19.linuxfoundation.org/wp-content/uploads/2017/11/When-eBPF-Meets-FUSE-Improving-Performance-of-User-File-Systems-Ashish-Bijlani-Georgia-Tech.pdf. Accessed: 09/04/2021.
[28] Xdp express data path. https://www.iovisor.org/technology/xdp. Accessed: 09/04/2021.
[29] Xdp performance regression due to config_retpoline spectre v2. http://lkml.iu.edu/hypermail/linux/kernel/1804.1/05171.html. Accessed: 09/04/2021.
[30] Supporting Linux kernel development in Rust. https://lwn.net/Articles/829858/, 2020. Accessed: 09/04/2021.
[31] Nadav Amit, Fred Jacobs, and Michael Wei. JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019.
[32] Nadav Amit and Michael Wei. The Design and Implementation of Hyperupcalls. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018.
[33] Maximilian Bachl, Joachim Fabini, and Tanja Zseby. A flow-based IDS using Machine Learning in eBPF. *CoRR*, abs/2102.09980, 2021.
[34] Sabur Baidya, Yan Chen, and Marco Levorato. eBPF-based content and computation-aware communication for real-time edge computing. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops, INFOCOM Workshops 2018, Honolulu, HI, USA, April 15-19, 2018*. IEEE, 2018.
[35] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture. In *Proceedings of the ACM SIGCOMM 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 30 - September 3, 1999, Cambridge, Massachusetts, USA*. ACM, 1999.
[36] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich. Efficiently Mitigating Transient Execution Attacks using the Unmapped Speculation Contract. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020.
[37] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating Linux Security with eBPF iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*. ACM, 2018.
[38] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of Defenses against Transient-Execution Attacks. In *GLSVLSI '20: Great Lakes Symposium on VLSI 2020, Virtual Event, China, September 7-9, 2020*. ACM, 2020.
[39] Cyril Renaud Cassagnes, Lucian Trestioreanu, Clément Joly, and Radu State. The rise of eBPF for non-intrusive performance monitoring. In *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 2020.
[40] YoungEun Choe, Jun-Sik Shin, Seunghyung Lee, and JongWon Kim. eBPF/XDP Based Network Traffic Visualization and DoS Mitigation for Intelligent Service Protection. In *Advances in Internet, Data and Web Technologies, The 8th International Conference on Emerging Internet, Data and Web Technologies, EIDWT 2020, Kitakyushu, Japan. 24-26 February 2020*, volume 47 of *Lecture Notes on Data Engineering and Communications Technologies*. Springer, 2020.
[41] Jonathan Corbet. KRSI — the other BPF security module. https://lwn.net/Articles/808048/, 2019. Accessed: 09/04/2021.

[42] Luca Deri, Samuele Sabella, and Simone Mainardi. Combining System Visibility and Security Using eBPF. In *Proceedings of the Third Italian Conference on Cyber Security, Pisa, Italy, February 13-15, 2019*, volume 2315 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[43] Victor Duta, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. PIBE: practical kernel control-flow hardening with profile-guided indirect branch elimination. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 2021.

[44] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. Synthesizing JIT Compilers for In-Kernel DSLs. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*. Springer, 2020.

[45] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019.

[46] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 2020.

[47] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In Xenofontas A. Dimitropoulos, Alberto Dainotti, Laurent Vanbever, and Theophilus Benson, editors, *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, pages 54–66. ACM, 2018.

[48] Jibum Hong, Seyeon Jeong, Jae-Hyoung Yoo, and James Won-Ki Hong. Design and Implementation of eBPF-based Virtual TAP for Inter-VM Traffic Monitoring. In *14th International Conference on Network and Service Management, CNSM 2018, Rome, Italy, November 5-9, 2018*. IEEE Computer Society, 2018.

[49] Clement Joly and François Serman. Evaluation of tail call costs in eBPF. *Linux Plumbers Conference 2020*, 2020.

[50] Taesoo Kim and Nickolai Zeldovich. Practical and Effective Sandboxing for Non-root Users. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. USENIX Association, 2013.

[51] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018.

[52] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby C. Murray, and Gernot Heiser. Formally verified software in the real world. *Commun. ACM*, 61(10), 2018.

[53] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, 2009.

[54] 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. Spectre Attacks: Exploiting Speculative Execution, 2019.

[55] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020.

[56] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. Safe and Efficient Remote Application Code Execution on Disaggregated NVM Storage with eBPF. *CoRR*, abs/2002.11528, 2020.

[57] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. SPEAKER: Split-Phase Execution of Application Containers. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, volume 10327 of *Lecture Notes in Computer Science*. Springer, 2017.

[58] Jialin Li, Samantha Miller, Danyang Zhuo, Ang Chen, Jon Howell, and Thomas Anderson. An incremental path towards a safer OS kernel. In *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*. ACM, 2021.

[59] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: reading kernel memory from user space. *Commun. ACM*, 63(6), 2020.

[60] Chang Liu, Zhengong Cai, Bingshen Wang, Zhimin Tang, and Jiaxu Liu. A protocol-independent container network observability analysis system based on eBPF. In *26th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2020, Hong Kong, December 2-4, 2020*. IEEE, 2020.

[61] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*. USENIX Association, 1993.

[62] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 2019.

[63] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020.

[64] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of Linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 2019.

[65] Elena Reshetova, Filippo Bonazzi, and N Asokan. Randomization can't stop bpf jit spray. In *International Conference on Network and System Security*. Springer, 2017.

[66] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. Performance Implications of Packet Filtering with Linux eBPF. In *30th International Teletraffic Congress, ITC 2018, Vienna, Austria, September 3-7, 2018 - Volume 1*. IEEE, 2018.

[67] Christopher Small and Margo Seltzer. VINO: An Integrated Platform for Operating System and Database Research, 1994.

[68] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. Demo/poster abstract: Efficient and flexible packet tracing for virtualized networks using eBPF. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops, INFOCOM Workshops 2018, Honolulu, HI, USA, April 15-19, 2018*. IEEE, 2018.

[69] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November*

4-6, 2020. USENIX Association, 2020.

[70] Dave Jing Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Peter C. Johnson, and Kevin R. B. Butler. LBM: A Security Framework for Peripherals within the Linux Kernel. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019.

[71] Marcos Augusto M. Vieira, Matheus S. Castanho, Racyus Pacifico, Elerson Rubens da Silva Santos, Eduardo P. M. Câmara Júnior, and Luiz Filipe M. Vieira. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.*, 53(1), 2020.

[72] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. USENIX Association, 2014.

[73] Zhenyu Wu, Mengjun Xie, and Haining Wang. Swift: A Fast Dynamic Packet Filter. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. USENIX Association, 2008.

[74] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. Leveraging eBPF for programmable network functions with IPv6 segment routing. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*. ACM, 2018.

[75] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing Safe and Efficient Kernel Extensions for Packet Processing. *CoRR*, abs/2103.00022, 2021.

[76] Cliff Young, Nicholas C. Gloy, and Michael D. Smith. A Comparative Analysis of Schemes for Correlated Branch Prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*. ACM, 1995.

[77] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 479–494. USENIX Association, 2021.

[78] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. BPF for storage: an exokernel-inspired approach. In *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*. ACM, 2021.