BeeBox: Hardening BPF against Transient Execution Attacks

Di Jin
Brown University

Alexander J. Gaidis
Brown University

Vasileios P. Kemerlis
Brown University

Abstract

The Berkeley Packet Filter (BPF) has emerged as the de-facto standard for carrying out safe and performant, user-specified computation(s) in kernel space. However, BPF also increases the attack surface of the OS kernel disproportionately, especially under the presence of transient execution vulnerabilities. In this work, we present BeeBox: a new security architecture that hardens BPF against transient execution attacks, allowing the OS kernel to expose eBPF functionality to unprivileged users and applications. At a high level, BeeBox sandboxes the BPF runtime against speculative code execution in an SFI-like manner. Moreover, by using a combination of static analyses and domain-specific properties, BeeBox selectively elides enforcement checks, improving performance without sacrificing security. We implemented a prototype of BeeBox for the Linux kernel that supports popular features of eBPF (e.g., BPF maps and helper functions), and evaluated it both in terms of effectiveness and performance, demonstrating resilience against prevalent transient execution attacks (i.e., Spectre-PHT and Spectre-STL) with low overhead. On average, BeeBox incurs 20% overhead in the Katran benchmark, while the current mitigations of Linux incur 112% overhead. Lastly, BeeBox exhibits less than 1% throughput degradation in end-to-end, real-world settings that include `seccomp-BPF` and packet filtering.

1 Introduction

The Berkeley Packet Filter (BPF) [9, 74] is increasingly gaining traction as an in-kernel, *"universal" virtual machine* (VM): that is, a generic mechanism for extending or customizing OS kernel functionality by "pushing" (i.e., performing) user-specified computation(s) in kernel space, in a safe and performant manner. BPF greatly improves the efficiency of certain tasks by moving application logic closer to where data is generated, processed, or received, thereby reducing kernel-to-user I/O and avoiding expensive context switches. It enables a plethora of mature and emerging applications, including security monitoring and policy enforcement [23, 69, 96], high-performance networking [36, 95], observability and tracing [46], storage [104], as well as scheduling [39]. More importantly, many BPF applications require, or can benefit from, allowing the use of the BPF infrastructure in an *unprivileged* manner [19]. Unfortunately, however, BPF increases the OS attack surface [50], and it has been extensively used as the underlying apparatus for mounting *transient execution attacks* [11] against the kernel (from userland).

Transient execution attacks abuse the speculative-code execution capabilities built into modern CPUs, effectively leaking data through (forced) mis-speculation and side channels [33, 71]. BPF facilitates such attacks [55, 56] as BPF code can be highly customizable and is executed in the same context as the OS kernel itself, hence bypassing deployed defenses [32, 41]. Modern OSes place a wide range of *security-sensitive* data in kernel space, such as cryptographic material [63], the memory contents of user processes (i.e., `physmap`/direct map) [53], as well as (secret) metadata that impact the effectiveness of system-wide security protection mechanisms, like (K)ASLR [25]. Hence, it is of paramount importance to mitigate such BPF-based information leakage, especially in the presence of unprivileged BPF.

The Linux kernel currently deploys a combination of mitigations [58] against BPF-assisted, transient execution attacks, including: (1) selectively blocking speculative code execution; (2) limiting pointer arithmetic operations; and (3) verifying the safety of all branch-instruction combinations, even when some of them are impossible. As we discuss in Section 3, these mitigations primarily extend the BPF code verification and interpretation/JITing process via a collection of *ad-hoc* checks. This approach of progressively *retrofitting* security to the BPF infrastructure has caused a range of problems, in terms of usability and performance, without providing clear protection guarantees [8, 91, 92]—and, as expected, it has also led to various bypasses [7, 29, 57]. For these reasons, Linux distributions *restrict* eBPF to privileged users only [19]. In this work, we propose BeeBox: a new BPF security architecture that protects against transient execution attacks through a principled, security-first design approach.

The main idea behind BeeBox is to *sandbox* the BPF data in an SFI-like (software fault isolation) [99] manner. Moreover, by using static analysis and various domain-specific properties, BeeBox elides enforcement checks to improve performance, without sacrificing security. BeeBox improves the security posture of BPF, thereby allowing unprivileged users to *safely* reap the benefits of the BPF (sub)system and utilize it to its full potential.

In summary, we make the following contributions:

- We present the design of BeeBox, a security architecture against challenging and prevalent transient execution attacks in BPF—namely, Spectre-PHT and Spectre-STL [56]—, which isolates sensitive data from BPF's speculative execution reach. Our design also covers the security of BPF helper functions and cBPF.
- We provide a prototype implementation of BeeBox, which supports popular BPF applications, as well as a collection of tools and methods that simplify the future expansion of BeeBox's supported features.
- We evaluate our prototype on synthetic and real-world BPF programs and workloads, demonstrating that BeeBox provides increased security, compatibility, and efficiency than Linux's current mitigations. BeeBox achieves 0–23% overhead in kernel micro-benchmarks, and $< 1\%$ overhead in end-to-end, real-world settings.

2 Background

2.1 Kernel Security

OS kernels are usually the most privileged piece of software on a system, making them a lucrative target for attackers. Unfortunately, OS kernels are typically written in memory- and/or type-unsafe languages, which are prone to memory errors [97]. An attacker armed with a memory corruption vulnerability in the kernel can tamper with control data (e.g., code pointers [59]) to hijack the kernel's control flow and eventually gain arbitrary code execution [54, 81]. The current, predominant technique for achieving arbitrary code execution is *code reuse* [10], where an attacker chains together snippets of existing (i.e., benign) code "out-of-context."

Several defense strategies have been proposed to mitigate code-reuse attacks, with the most popular relating to information hiding or integrity checking. Schemes of the former attempt to conceal sensitive information by either randomizing [25] or diversifying [26, 60] the layout of memory, or by employing new memory protection policies, such as execute-only memory (XOM) [83, 89]. Schemes of the latter attempt to ensure that computed control-flow transfers are valid according to a given policy [5, 28, 59, 61]. Another technique for defending against code-reuse attacks in the kernel relies on isolating kernel memory from the attack payload, since real-world exploits may require complex payloads that do not fit into a limited corrupted region. Defenses in this direction prevent the kernel from using attacker-controlled payloads in user space [20, 54, 103] or in kernel memory regions that are effectively user-controlled [50, 53].

Privilege escalation can also be achieved by *data-only* attacks [38, 47], where, instead of control-flow data, the attacker overwrites targeted data and escalates privilege without triggering any control-flow integrity violations. Defenses against generic data-only attacks, including DFI (data-flow integrity) [16] and full memory safety [76, 77], remain impractical for performance reasons. Selective isolation schemes such as xMP [84] and ISLAB [75] were proposed to protect critical data such as page tables, process credentials, and memory management metadata.

Apart from privilege escalation, another goal of kernel attacker is *information disclosure*, which can be leveraged to obtain sensitive information managed by the kernel, such as cryptographic keys or secrets of other processes [53]. However, exploiting memory vulnerabilities in software is not the only way to achieve information disclosure; hardware vulnerabilities, like Meltdown [56] or cache-based side channels [33, 71], are an effective, alternate route.

2.2 Berkeley Packet Filter

Overview. BPF [74] was originally designed to accelerate network monitoring by performing user-defined packet filtering in-kernel rather than in user space. Since its inception, BPF has grown into a generic facility in Linux that acts as a *universal*, in-kernel VM for user-defined computation [18].

While BPF continues to support efficient networking applications, such as packet filtering [31] and load balancing [80], a variety of other applications capitalize on it to increase security and support new functionality. Several popular applications, such as Docker, Firefox, Chromium, and OpenSSH, employ `seccomp-BPF` [69] to filter system calls (syscalls) according to user-defined policies, with recent research exploring automatic generation [23] and secure enforcement [27] of syscall policies. Additionally, BPF underpins many tracing tools [9], since users can attach BPF programs to Kprobes (kernel probes) [93], granting increased visibility into the workings of a system. Other use cases for BPF include reducing context switching overhead in FUSE (Filesystem in USErspace) [6] and improving throughput and latency of the kernel's storage stack [104].

Design. When a BPF program is pushed into the kernel, it is first verified for safety and then attached to a kernel hook point. Later, when a requisite event fires (e.g., a syscall is made or a packet is received), the installed program is executed. Linux currently supports two types of BPF: *classic BPF* (cBPF) and its more feature-rich successor, *extended BPF* (eBPF). The typical use cases for cBPF involve syscall [69] and packet filtering [31]. The cBPF machine abstraction consists of two 32-bit registers and 64 bytes of addressable scratch memory.

cBPF code is translated internally to eBPF, allowing the kernel to maintain a single BPF runtime. eBPF greatly expands upon cBPF's capabilities and use-cases, offering: 10 general-purpose, 64-bit registers; a read-only frame-pointer register; and up to 512 bytes of addressable scratch memory (i.e., stack space). The instruction set of both cBPF and eBPF allows for memory loads and stores, moving values between registers, ALU operations, and branching. However, eBPF additionally provides the ability to call kernel-native functions, dubbed *helpers* [1], from within eBPF programs. Helpers provide various utilities and help maintain BPF *maps*—a variety of data structures implemented atop generic, kernel-resident key/value stores that allow eBPF programs to persist state across invocations and expose data to userland. All BPF programs also have *types*, which inform the kernel where the programs can be attached, what helpers and maps can be accessed, and what *context* (i.e., arguments and data) should be provided with upon invocation [68]. For example, an eBPF program with type `xdp` (`BPF_PROG_TYPE_XDP`) will run on a network device and the context passed to it will consist of pointers to the raw packet data, which can be used to perform packet rewriting and forwarding.

Safety and Security. BPF is designed to be safe to run within kernel context. To achieve that, when BPF programs are loaded into the kernel they pass through a static *verifier*, which is guaranteed to be sound but not complete—i.e., a safe program may be rejected, but an unsafe program will not pass. The BPF verifier ensures similar properties for both cBPF and eBPF, but verifies them separately, performing a much simpler analysis for cBPF programs. For eBPF, the verifier mainly verifies three properties: (1) program termination; (2) memory safety; and (3) type safety when calling helpers. For (1), the verifier simply ensures that all branching instructions only "jump forward." For (2) and (3), the verifier uses static analysis techniques to determine the register types, whether they are pointers or scalars, *etc.* For a scalar, the analysis tries to figure out the range of its possible values; for a pointer, the analysis needs to determine what kind of memory it points to and whether it points within bounds. Further, the BPF verifier is also in charge of rewriting BPF program code to adjust helper-call targets and inline various helpers as BPF instructions when possible. Unfortunately, while performing static analysis to avoid runtime checks is crucial for the performance of BPF, the verifier is known to be error prone [62] and susceptible to speculative execution attacks (§3).

Support. Both LLVM and GCC maintain eBPF backends, which compile programs written in C-like syntax to eBPF ELF files. Additionally, both of these BPF compilers—which are different from the in-kernel (BPF) JIT compiler—provide various utilities for working with eBPF ELFs. In a similar vein, `libbpf` [2] is a C library maintained by the Linux kernel which provides a low-level API for loading eBPF ELFs and managing eBPF programs, maps, and events from userland.

2.3 Transient Execution Attacks

Modern CPUs maximize their performance by employing various techniques to avoid CPU idling, such as *out-of-order execution* and *speculative execution* [35]. Instructions that are executed out-of-order, or speculatively, but never committed to the CPU's architectural state are considered to be *transiently executed*. Although transiently executed instructions are never architecturally visible, they can leave observable side effects in the CPU's micro-architectural state (e.g., the cache [56], load ports [90], line-fill buffers [98], store buffers [14]). This has enabled *transient execution attacks*, where an adversary coerces the CPU into transiently accessing unintended information before using micro-architectural side channels [33, 71, 98, 100, 102] to leak information.

The Spectre family of attacks is one class of transient execution attacks that abuse CPU predictors, via *(mis)training* or *tampering*, to trigger speculative execution of instructions that access sensitive data, which can later be inferred through side channels. There are a handful of Spectre variants, classified by the CPU predictor they target. Spectre-PHT (Spectre-v1) [11] targets the pattern history table (PHT)—used to predict the outcome of conditional branches—to trigger the speculative execution of instructions preceded by a conditional branch. Spectre-BTB (Spectre-v2) [56] targets the indirect branch predictor, i.e., the branch target buffer (BTB), to trigger the speculative execution of forward-edge indirect branches. Spectre-STL (Spectre-v4) [48] abuses the memory disambiguator to load data before a prior, dependent store completes.

A variety of defenses have been proposed to address transient execution attacks, spanning both software [17] and hardware [37]. In the Linux kernel, the Spectre-PHT mitigation relies on developer annotations to identify when an array index is not trusted and add bit-wise operations to bound array indices during mis-speculation [65]. Spectre-BTB is mitigated via retpolines [43] and IBRS [41].

There are other root causes of transient execution, but predictor-based speculation and unexpected exceptions are the most prevalent and well-researched [85]. For exception-based transient execution attacks such as Meltdown [56] and Fallout [14], the problems are typically fixed through hardware changes [45], since it is generally clear what should happen in such situations: transient execution should respect potential exceptions. For predictor-based transient execution, isolation can be enforced such that the predictors cannot be influenced across contexts [41, 44] (e.g., influencing the kernel from userland), which can mitigate attacks that rely on abusing shared micro-architectural states [11, 72]. But for intra-context attacks, the safety of speculation is inherently software-defined, and solutions that prevent speculation, in general, are expensive [56]. Hence, selective application of various mitigations (such as `lfence` [40] or `SSBD` [13]) in targeted places is recommended by vendors [42] and has begun to receive attention by researchers [24].

3 Motivation

The combination of BPF and Spectre attacks allows an adversary to mount transient execution attacks to leak sensitive data from the kernel with little noise and without reliance on software vulnerabilities [55,56]. As a result, eBPF functionality is disabled by default for unprivileged users [19,87,94]. Features that use cBPF, such as `seccomp-BPF`, have seen proposals to support eBPF, but were rejected due to the complexity and difficulties that stem from maintaining certain mitigations [51]. Yet, the full set of BPF functionality is desirable for a lot of applications. Apart from `seccomp-BPF`, there are a plethora of proposed applications that try to use BPF in an unprivileged manner, including high-speed storage systems [104] and scheduling [39,52]. XDP-based applications [36], such as Cillium [95] and Katran [80], would also benefit from a reduction in privilege required by the BPF runtime.

Currently, if unprivileged BPF is enabled, the BPF verifier and JIT engine are responsible for applying BPF-specific Spectre mitigations at load time [58], which we collectively refer to as *Linux provisional mitigations* (LPM). For Spectre-PHT, the verifier rewrites pointer arithmetic to bound results within a given object, similar to the `array_index_nospec` macro [65]. To protect type safety under Spectre-PHT, the verifier's static analysis enforces type consistency across all combinations of branch choices, even impossible ones (e.g., a path where two branches with contradicting conditions are both taken). For Spectre-BTB, the JIT engine adds instrumentation for indirect branches in accordance with the kernel-wide defenses (such as retpolines [43]). For Spectre-STL, the verifier emits speculation barriers (e.g., `lfence` [40]) after stores to the stack, whenever pointers are involved.

A summary of defenses the Linux kernel deploys against transient execution attacks is provided in Table 1. Spectre-BTB and Meltdown are mitigated by generic, kernel-wide defenses, such as retpolines [43], IBRS [41], and KPTI [66], while generic Spectre-PHT and Spectre-STL defenses, such as SLH [15,82] and SSBD [13], are expensive to apply to the whole kernel. Consequently, the LPM defend against these attacks specifically in the context of BPF; however, the LPM have various issues regarding *compatibility*, *complexity*, *scope*, and *performance*, which we summarize below.

Compatibility. The LPM introduce additional analysis complexity and safety requirements to the verifier, causing it to reject (previously) safe code patterns in BPF programs. For example, Cillium [95] breaks because it performs variable-offset stack accesses, and Katran [80] breaks because it requires conditional pointer arithmetic. In addition, valid BPF programs experience verifier state explosion due to the verification regarding impossible branch combinations (§7.1.3). This places a burden on application developers to either address the verifier's requirements by rewriting code patterns targeted by the mitigations, or decide to forgo protection and load the program with elevated privilege.

Complexity and Scope. The LPM were formed over time, in an ad-hoc manner, as a response to repeated bypasses of Spectre-PHT and Spectre-STL mitigations caused by incompleteness and/or corner cases [7,29,57]. Progressively retrofitting security in this manner adds complexity to the BPF (sub)system that makes it difficult for kernel developers to reason about the safety of interactions between proposed verifier changes and the LPM [8,91,92]. Further, despite all this added complexity, the mitigations still provide incomplete protection, not covering cBPF or native BPF helper functions.

Performance. The mitigation against Spectre-STL inserts speculation barriers (e.g., `lfence`) after any pointer spill, and subsequent load, targeting the same stack slot. As a result, the LPM incur significant runtime overhead, especially in non-trivial BPF programs (§7.2.2). In general, it is difficult to stop mis-speculation in a way that does not hinder performance [12], which is the root cause of complexity in LPM.

4 Threat Model

Adversarial Capabilities. We assume an attacker that has access to BPF functionality and seeks to disclose sensitive information from the kernel. The attacker also has the appropriate rights to use the desired types of BPF programs (e.g., `cap_net_admin` for `XDP`), but is otherwise unprivileged. We do not assume the attacker can install BPF programs with `cap_perfmon`, since this capability can disclose a wide range of sensitive values already (such as register values at any point) [67]. As a result, performance-related BPF types are not available to the attacker, like the BPF programs that attach to Kprobes [93] and tracepoints [9].

As an unprivileged user of the system, the attacker can create any BPF program in the kernel that passes verification, as well as control the contents of BPF maps within allowed semantics. The attacker is also assumed to have the ability to arbitrarily trigger speculation at any point between the entry and exit of their BPF program, including when it invokes helper functions—these are only limited by potential speculation barriers. Under such speculative execution scenarios, the attacker can also arbitrarily flip conditional jumps and skip or forward any store to any load targeting the same address.

Hardening Assumptions. Regarding the kernel, we assume that the concrete execution of any kernel or BPF-runtime code is safe, and that there are no errors in the BPF verifier or JIT engine. Any execution of kernel code, concrete or speculative, before and after execution of a BPF program, is also assumed to be safe—i.e., the kernel does not leak any sensitive information into BPF programs' speculation, nor does it incorrectly use unsafe values after BPF programs' speculation. The only defense against Spectre-PHT and Spectre-STL for BPF is BeeBox. Further, we assume that the kernel has appropriate defenses, in either software or hardware, against other cross-privilege transient execution attacks, such as Meltdown [56].

Category	Feature	LPM [58]	retpoline [43]	IBRS [41]	KPTI [66]	BeeBox
Security	Block Spectre-PHT in BPF code	✓				✓
	Block Spectre-PHT in BPF helpers					✓
	Block Spectre-STL in BPF code	✓				✓
	Block Spectre-STL in BPF helpers					✓
	Block Spectre-BTB		✓	✓		
	Block Meltdown				✓	
Compatibility	Allow conditional ptr. arithmetic in unpriv. BPF					✓
	Avoid verifier state explosion in unpriv. BPF					✓

Table 1: Existing Linux kernel defenses and BeeBox's coverage over transient execution attacks and compatibility features.

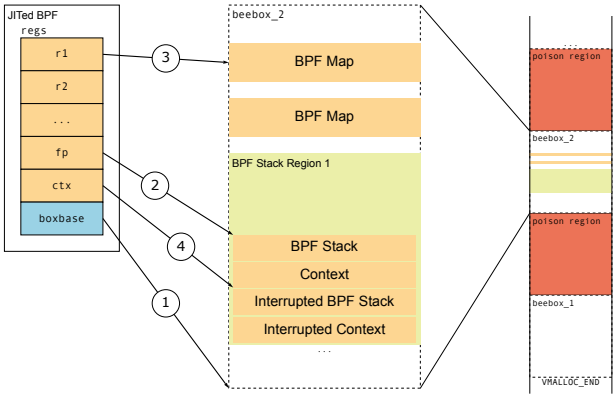


Figure 1: The memory layout of BeeBox. The left part represents the CPU register states when a JITed BPF program is running; the middle part is a magnified view of the memory layout of a `beebox`; and the right part represents the virtual address space, where `beebox`es are placed after the `VMALLOC` region, with a 4GB poison region in-between adjacent pairs.

Defenses against memory errors, such as control-flow integrity [5], page-table protection [22], pointer integrity [59], and data isolation schemes [75,84] are orthogonal to BeeBox.

5 Design

The goal of BeeBox is to prevent BPF programs from accessing sensitive data, even under speculative execution. Fortunately, the majority of data that BPF programs access (e.g., the BPF stack and BPF maps) are designed to be visible to users. This key observation allows BeeBox to identify what data should be isolated, which does so by leveraging SFI techniques [99], tailored to the speculative execution domain. Specifically, BeeBox places all data a given BPF program will access inside a dedicated memory region, called `beebox`, and applies SFI-style isolation to the program, confining all of its memory accesses to this region. BeeBox thus ensures memory safety, as the BPF program can only access data inside the `beebox`—concretely or speculatively—, which does not contain any sensitive data.

5.1 BeeBox Sub-address Space

BeeBox isolates data directly accessed by BPF programs in `beebox`es, as shown in Figure 1. A `beebox` is created for each user, such that no information can be leaked to, or from, other users (including user `root`). For example, as shown on the right of Figure 1, `beebox_1` and `beebox_2` are two separate `beebox`es that were created for two different users. In addition, 4GB poison zones are placed between each `beebox` to support optimizations (§5.4). The kinds of memory objects placed inside each `beebox` sub-address space are magnified in the middle portion of Figure 1 and described below.

BPF Stack. The BPF stack is a region where BPF programs can temporarily store intermediate computation results that are not semantically shared with other components (e.g., the kernel itself or user applications). Since BPF programs are executed with {CPU, core}-migrations disabled—i.e., the BPF program invocation will stay on the same CPU—, we allocate a BPF stack region for each CPU in a `beebox`.

BPF Maps. BPF maps hold user-owned data, which can be accessed by both BPF programs and their corresponding user process (via the `bpf` system call). To support maps securely, their defining data structures are split: map data and some metadata essential to map operations (e.g., lookup and update) are safely placed in the `beebox`, since their contents are either controlled or can be inferred by the user; the remaining metadata used by the kernel itself (e.g., associated function pointers) remain outside of the `beebox`. Some metadata, such as the map size, are duplicated for the kernel code to access, because their copy in `beebox` is untrusted under speculation.

BPF Context. Handling contexts poses unique challenges since context data structures can interleave private kernel data and BPF-accessible data. For example, a BPF program of type `BPF_PROG_TYPE_SOCKET_FILTER` will receive a `struct sk_buff` in its context that has a large number of fields, out of which only a handful are made accessible to the BPF program itself (via `struct __sk_buff`). For these BPF program types, BeeBox copies the relevant parts of the context data structure that are visible to the BPF program into the BPF stack region. In cases where private kernel data is not interleaved with BPF-accessible data, the context data structure is usually generated and filled on the kernel's stack, just before the BPF program's invocation (as in `seccomp-BPF`).

Here, the non-interleaved context can be generated on the BPF stack region and no other changes are required. Network-related BPF programs usually have access to a packet buffer, and a pointer to a packet, which, in this case, is part of the context object. This is a common pattern in BPF, and the packet can be seen as an extension to the context. Through accurate static analysis done by the BPF verifier, accesses to these packets are identified and protected against overflowing. We handle packets the same way we handle context: we copy them into `beebox`. Note that copying context data structures (and packets) can be costly; we designed certain optimizations that can be applied to different types of BPF programs (§5.4).

Space Considerations. The size of the `beebox` region is 4GB. We chose this size mainly to allow for efficient SFI instrumentation (§5.2). Additionally, the stack region of each CPU reserved for BPF programs should be $(c+p)\cdot l+s$, where: c is the size of the largest context, p is the size of the largest packet we allow BPF to handle, l is the level of nested interrupts which can have BPF running, and s is the size of the kernel stack. This equation stems from the fact that all BPF stack frames are allocated on the kernel stack in the vanilla Linux kernel, and each nested interrupt can have at most $c+p$ bytes added on top of that. We have chosen the total reserved area per-CPU to be 280KB to cover $p = 64$KB, $c = 4$KB, $l = 4$, and $s = 8$KB. (These sub-regions are allocated and mapped during bootstrap, so they cannot be arbitrarily sized.)

5.2 Securing BPF Programs

Instrumentation. BeeBox uses SFI-style isolation to ensure pointers used by a BPF program are confined to a 4GB `beebox` region that contains only user-accessible data. Pointers embedded in a BPF program during JITing, and pointers passed to the program by the calling context, are transformed into `boxptrs`: 32-bit offsets within the `beebox` region. When a memory access needs to be performed, a usable 64-bit pointer is created from a given `boxptr` by clearing the upper 32 bits of the `boxptr` and adding the result to a register—the `boxbase` register—, which contains the starting address of the `beebox` region (①in Figure 1). To initialize the `boxbase` register, the JIT engine emits a native instruction at the beginning of a BPF program which loads the base pointer, encoded as an immediate value, into the `boxbase` register. The `boxbase` register is never spilled or used in any other way, ensuring it cannot be controlled speculatively (or leaked); BeeBox guarantees the target of memory accesses from within BPF programs are always confined inside the `beebox` region. (Figure 3 shows an example of this transformation in x86-64.)

BPF Stack. There is a per-CPU pointer keeping track of the current top of the BPF stack allocated by BeeBox. The pointer is updated by locally atomic instructions to be safe against interrupts, since BPF programs can run with interrupts enabled and can also be invoked within interrupt context.

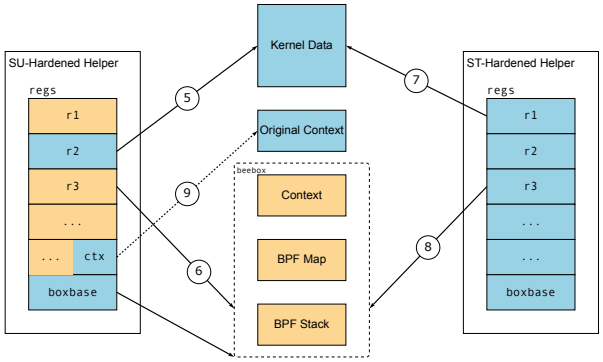


Figure 2: Memory layout and CPU states when helper functions are executed. Blue indicates the value is speculatively trusted; yellow indicates otherwise. The left part represents the CPU state when running an SU-hardened helper; the right part corresponds to running an ST-hardened helper.

BeeBox modifies the JIT compiler to emit inlined prologues and epilogues to move the BPF stack pointer and set the frame pointer register accordingly (②in Figure 1).

BPF Maps. When BPF maps are accessed by the kernel on behalf of the user's query via system calls, the kernel uses normal pointers to dereference the map data inside a `beebox` (instead of `boxptrs`). However, values loaded from the `beebox` speculatively should not be trusted and should be handled safely (e.g., masking the value before using it), which is consistent with the kernel's current security model regarding user input [70]. When BPF maps are accessed directly from BPF programs (e.g., ③in Figure 1), BeeBox uses 32-bit `boxptrs`, and the metadata is either embedded in the JITed program or looked up from the copy inside `beebox`. When BPF maps are accessed by helpers, the BPF program passes `boxptrs` to (modified) helper functions which use them safely.

BPF Context. Regardless of whether private kernel data is interleaved with BPF accessible data, BeeBox transforms the context pointer (④in Figure 1) into a `boxptr` at the callsite of the kernel that invokes a given BPF program. If the context includes a packet, the packet is also copied into the `beebox`. All pointers in the context data structure that point into the packet are also converted to `boxptrs`.

5.3 Securing Helper Functions

There are two hardening strategies for securing BPF helper functions. *SU-hardening* (Speculatively Untrusted, §5.3.2) protects helper functions such that they can be safely run during (untrusted) speculation. *ST-hardening* (Speculatively Trusted, §5.3.3) ensures that helper functions are run under the precondition that there is no influence from untrusted speculation. Figure 2 illustrates the memory layout and CPU states when helper functions are executed under both schemes.

```
1  bpf_get_current_uid_gid:
2    mov    rax, QWORD PTR gs:0x1ad00  # struct task_struct
3    test   rax, rax
4    je     1f
5    mov    rax, QWORD PTR [rax+0x638] # struct cred
6    mov    rax, QWORD PTR [rax+0x4]   # uid, gid
7    ret
8  1:
9    mov    rax, 0xffffffffffffffea    # -EINVAL
10   ret
```

Listing 1: A compiled helper function, `bpf_get_current_uid_gid`, where all memory loads are via clean pointers.

5.3.1 Clean Pointers

We first introduce the definition of *clean pointers*, an important concept for our discussion later. All memory accesses using `boxptr`s can be confined with BeeBox's SFI scheme (§5.2); however, certain BPF helpers use native pointers that point outside of a `beebox`, and hence cannot be transformed into `boxptr`s. These native pointers can potentially be speculatively-hijacked by the BPF program, escaping BeeBox's defense. Since we assume the correctness of the helper functions' concrete execution, we can avoid speculatively targeting unsafe locations by placing speculation barriers (e.g., `lfence` instructions) before native pointer dereferences. This naïve solution can be greatly improved upon with a key insight: native pointers in helper functions that are uninfluenced by any untrusted input (e.g., arguments passed from BPF programs) do not require a preceding speculation barrier—they can be safely dereferenced without leaking sensitive kernel data. We consider such pointers as *clean*.

Clean pointers are identified by static taint analysis, where the sources of taint are untrusted input (e.g., helper function arguments, values loaded from the `beebox` region, values loaded from the stack). If a native pointer is not tainted when dereferenced, then the dereference is considered to be done via a clean pointer. (`boxbase` is handled like a clean pointer.) Importantly, clean pointers are native pointers, so their values must be prevented from leaking into attacker visible realms (e.g., `beebox` memory and the branch predictor). Also, clean pointers are read-only, because any write to a kernel address can contaminate future reads to that address (as specified by our threat model), which may compromise the "cleanliness" of other (clean) pointers derived from the address.

Listing 1 shows a BPF helper function, `bpf_get_current_uid_gid`, which is used for getting the current task's user ID (UID) and group ID (GID). This function uses three native pointers to: get the current `struct task_struct` (line 2); extract its security context, `struct cred` (line 5); and read the task's UID and GID together, as a single quadword (line 6). Since all memory accesses in the function refer to private kernel data, and since all pointers are uninfluenced by BPF program data, all three native pointers are considered clean pointers and do not require any hardening from BeeBox.

5.3.2 Speculatively Untrusted Hardening

Speculatively untrusted hardening (SU-hardening) assumes that the execution of BPF helper functions can be run under untrusted speculation. Given this, BeeBox ensures that: (1) all memory accesses into the `beebox` by the helper are SFI-hardened (⑥ in Figure 2); (2) no native pointers are leaked into the `beebox`; and (3) all memory accesses to external (i.e., kernel) data are either prefixed with a speculative barrier or accessed via a clean pointer (⑤ in Figure 2). As far as (2) goes, BeeBox modifies the helpers to store the local variables using the BPF stack, instead of the kernel stack, while keeping the return addresses outside of `beebox`. This is to prevent the attacker from mounting a Spectre-STL against stale kernel stack values through the helper.

Since SU-hardening requires speculation barriers prior to external memory accesses that are not through clean pointers, this technique is better suited to BPF helper functions that do not access lots of external data. For example, certain utility helpers (e.g., `bpf_strtol`) and the majority of BPF map helpers only operate on information already in the `beebox`. However, some helpers that do access external data mainly through clean pointers (e.g., `bpf_get_smp_processor_id`) do benefit from this approach as well.

SU-hardening is achieved through three main steps. First, the interface between BPF programs and helper functions is modified to replace native pointers with `boxptr`s, which maintain their 32-bit representation throughout the helper's execution. Since `boxptr`s cannot be accessed directly, they go through a special C macro, `unbox`, which applies SFI instrumentation and makes the `boxptr`s usable. (The `unbox` macro uses the `volatile` keyword to nail down the exact point of dereference in the compiled code and avoid spilling any native pointers before they are used.) Second, BeeBox ensures that helper functions maintain the "cleanliness" guarantee of the `boxbase` register by ensuring it is only used in the `unbox` macro and never spilled to the stack or overwritten. Finally, BeeBox ensures that external memory accesses are exclusively done via clean pointers, else they are prefixed with a speculation barrier. (This is currently done manually; a binary analysis tool based on our clean pointer identification algorithm is needed if one wishes to automatically ensure function is safe. We leave this for future work.)

5.3.3 Speculatively Trusted Hardening

Speculatively trusted hardening (ST-hardening) secures a helper by providing the guarantee that all speculation in a given BPF helper function will be safe, and thus no additional considerations are needed for accessing kernel data (⑦ in Figure 2). This hardening technique is fitted for helper functions that access kernel data much more than data produced by BPF programs. For example, `bpf_skb_clone_redirect` will clone the `sk_buff` passed in, which involves accessing a lot of kernel data, including the heap allocator's metadata.

To achieve this guarantee, BeeBox sanitizes any data flow originating from the calling context or the beebox. Thus, at a given helper's entry point, all boxptrs are converted into native pointers and a speculation barrier is inserted to avoid speculative execution on unsafe arguments. Additionally, accessing beebox memory requires a speculation barrier (⑧ in Figure 2), which is provided by another utility macro BeeBox defines. By default the speculation barriers are inserted; however, if a developer desires, they can (carefully) optimize-away the barriers. Finally, in the JITed BPF program, another speculation barrier is used to avoid unintended values from leaking back into the BPF's CPU state.

5.4 Performance Optimizations

We refer to the scheme combining all the techniques up until this point as *BeeBox-Basic*. It comprehensively stops any Spectre-PHT or Spectre-STL attack originating from unprivileged BPF programs; however, it can be costly, due to the copying of context and dense SFI-instrumentation. We address these problems in the following subsections.

5.4.1 Generic Optimizations

Reduced Copy. The first optimization variant, called *BeeBox-RC* (Reduced Copy), improves performance by reducing the amount of data that is copied into a beebox. Notably, this optimization strategy is applicable to every type of BPF program that does not have a stack-based context. At a high-level, by using the static analysis done by the verifier, a BPF program's accesses to contexts and packets are tracked and only the used portion of them is copied into the beebox.

BPF does not support variable-offset context accesses. Leveraging this fact, BeeBox-RC records up to 16 distinct context access offsets during the verifier's static analysis, and stores them along with the BPF program. If the number of accessed fields is within the limit, BeeBox-RC only copies each accessed field of the context data structure into the beebox.

For packets, because BPF supports variable-sized accesses, BeeBox-RC reuses the range analysis results of the max (packet) access offset, only copying-in the packet up to the size of the max offset that the BPF program will access. Because networking-related BPF programs typically operate on only the header of packets, the maximum size of packet copying is usually bounded to a reasonable number.

5.4.2 Case-specific Optimizations

Clean Context and Packet Pointers. The second optimization variant, called *BeeBox-CP* (Clean Pointer), makes the context pointer a clean pointer (i.e., a native pointer instead of a boxptr; ⑨ in Figure 2) to avoid copying contexts. Consequently, the context pointer is handled with the guarantees that clean pointers require: it is never spilled into memory and does not leak into other data flows—aside from being used as a base for dereferencing. As a result, the context pointer is treated similar to boxbase, where calls to helper functions avoid clobbering or spilling it. Other guarantees of the clean context pointer are checked in the verifier, and the optimization only applies when the context pointer is indeed clean. However, to ensure that BPF programs produced by a BPF compiler adhere to such requirements, and can hence utilize the optimization, BPF compiler changes are necessary. (For the purposes of this work, we manually produced compliant BPF programs in order to make use of the optimization.)

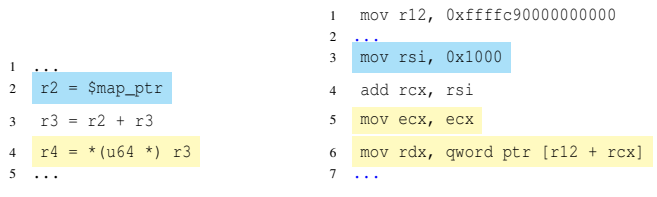
Importantly, since the context is a clean pointer pointing to the original context data structure outside of the beebox, the packet pointer enclosed inside the context is also a clean pointer. To safely allow variable-size accesses into the packet using clean pointers, BeeBox-CP (re)uses the static analysis done by the verifier to accurately identify and rewrite packet accesses into calls to custom helper functions that utilize the array_index_nospec macro in the kernel.

Ring-buffer No-copy. The third optimization variant applies when the type of the BPF program already requires the user to have network-device access privileges, e.g., xdp. In this case, we avoid copying packets by directly placing the device's ring buffer (or the packet it creates when the driver copies packets from the ring buffer into non-DMA memory, if the driver does not have support for XDP natively) in the beebox. Note that this optimization cannot work with higher-level, network-related BPF programs (such as socket-level BPF), because when such a packet is allocated, either at the driver or NAPI level, the destination socket is unknown to the kernel. We name this optimization variant *BeeBox-RB*.

Optimizing memcpy and Friends. In certain BPF helpers, functions such as memcpy and memcmp are used. These functions are simple but have extensive pointer dereferences, which seriously impact performance if SFI instrumentation is applied to all of them. We optimize this type of function by "unboxing" the arguments to native pointers when calling them, and manually ensuring that these native pointers are not changed or spilled in the function. Because the size of copying, or comparing, is a 32-bit integer value, we place a 4GB poison zone after the beebox region to confine all potential memory accesses by these functions.

6 Implementation

We implemented a prototype of BeeBox on x86-64 Linux v6.1. Due to the sheer number of features available to BPF, our prototype of BeeBox does not cover everything: we only BeeBox-hardened a subset of all the available features (of vanilla BPF) to support popular BPF applications. Our implementation consists of three inter-connected components, which we describe below.



(a) Vulnerable BPF program. (b) BeeBox JIT result.

Figure 3: Example of BeeBox instrumentation.

6.1 BeeBox-aware JIT Compiler

We reserve `beebox` regions in the unused part of the kernel's address space, after the `vmalloc` region, and we implemented a new allocation function, `beebox_alloc`, which wraps `__vmalloc_node_range` to provide custom ranges for allocating inside each `beebox`. In the JIT compiler, we added an instruction to load the base of the appropriate `beebox` into `r12`, which is not used by the current JIT compiler and acts as the `boxbase` pointer (e.g., line 1 in Figure 3b). We also modified the verifier to insert a new BPF (pseudo) instruction, `BPF_BOXMEM`, directly before load or store instructions in the BPF bytecode. `BPF_BOXMEM` marks the following instruction, signalling to the JIT process that the access should be SFI-hardened. During JITing, `BPF_BOXMEM`-ed memory accesses are transformed into two instructions that make up the core SFI instrumentation. First, a `mov` with a 32-bit register operand (e.g., `eax` for `rax`) is inserted to clear the upper 32 bits of the register (line 5 in Figure 3b). Then, a second `mov` instruction is inserted with a *base* + *index* addressing mode, where `r12` (i.e., the `boxbase` pointer) is used as the base and another 64-bit register, which stores the offset into the `beebox`, is used as the index (line 6 in Figure 3b). These two `mov`s bound memory accesses inside the 4GB `beebox` region.

In each `beebox` region, we allocate a 280KB stack for every CPU. Each BPF stack has a corresponding variable representing the top of the stack as a `boxptr`. We modified the JIT compiler's handling of function prologues and epilogues in order to "open" and "close" stack frames in the `beebox` stack region, instead of the normal stack, and to move the frame pointer into `ebp`, which is a 32-bit `boxptr`. All BPF stack accesses are frame-pointer-based so we avoid having to move `rsp` and handle additional stack swapping. Lastly, all stack accesses identified by the verifier are prepended with the `BPF_BOXMEM` BPF instruction, which are later correctly JIT-compiled to SFI-hardened native code.

6.2 Maps, Helpers, and Language Tricks

To simplify development, and debugging, when writing C code that interacts with pointers to a `beebox`, we added a special attribute, i.e., '`__beebox`', to annotate `boxptr`s in the kernel—the `__beebox` attribute is defined as `__attribute__((noderef))` when the source code is inspected by the static analysis tool `sparse` [64]. We chose to implement `boxptr`s as an attribute instead of defining a new type since the attribute can transparently keep the pointer's original type information. This makes incorrect propagation and accidental dereference noticeable at compile time, which streamlined the development of our prototype (otherwise, it would be difficult to apply the `unbox` macro correctly in expressions like: `&array_map->map->elem[i].lock`).

```c
struct bb_pcpu_freelist_head {
    struct bb_pcpu_freelist_node __beebox *first;
    raw_spinlock_t lock;
};

struct bb_pcpu_freelist {
    ...
    struct bb_pcpu_freelist_head extralist;
};

struct bpf_htab_inner {
    ...
    struct bucket __beebox *buckets;
    struct bb_pcpu_freelist freelist;
    u32 n_buckets;
    ...
};
```

Listing 2: Example of data structures placed inside `beebox`, which only contains information that can be exposed to the user. All pointer fields are annotated with '`__beebox`'.

We currently support five types of maps commonly used by BPF programs: *array map*, *per-CPU array map*, *array map of maps*, *hash map*, and *LRU hash map*. When data structures such as linked lists—which are needed for hash map—are moved into a `beebox`, their internal pointers are also converted into `boxptr`s. In Listing 2, we show how certain data structures placed in `beebox` are defined. All the embedded pointers are annotated with the `__beebox` attribute, and their value represents the offset within the `beebox`. We also duplicated and adapted the accessor macros, and inline functions, for these data structures to work with `boxptr`s.

We reserved `r12` when compiling helper functions to ensure it is never spilled or contaminated. In addition, we changed the function signature of the helpers we ported to BeeBox to use `boxptr`s; thus, together with the changes in the data structures, the compiled helper functions will not contain any accidental un-instrumented pointer dereferences. The counter-part to these helper functions is the call sites in the JITed BPF programs; they are transformed by the verifier to use `boxptr`s to point at the in-`beebox` part of the BPF maps.

Array map, per-CPU array map, and array map of maps each have three helper functions that need to be adjusted for the BPF runtime: we support two in array map, three in per-CPU array map, and one in array map of maps. For other helpers, we support everything used by the BPF programs in our evaluation, including $2/42$ common helpers, $6/14$ `socket`-specific helpers, and $1/29$ `xdp`-specific helpers.

6.3 Supported BPF Program Types

cBPF. We support cBPF in `seccomp-BPF` and packet filtering. Our analyses and BPF-level transformations are implemented in the cBPF verifier. The stack accesses in cBPF use dedicated cBPF instructions, hence we insert `BPF_BOXMEM` before the translated eBPF instruction(s). `seccomp-BPF`'s context is stack-based so it is just moved to the BPF stack. For packet filtering cBPF, we applied BeeBox-CP (§5.4.2), while packet accesses go through instrumented helpers.

Socket Filter eBPF. For eBPF-based socket filters, we implemented both BeeBox-RC (§5.4.1) and BeeBox-CP (§5.4.2). To implement BeeBox-RC, all stack accesses and map accesses remain the same; for context access, we insert `BPF_BOXMEM` before each context-memory access. We also changed the invocation of BPF programs to copy selected parts of the context and packets into the `beebox`, which we transformed accordingly. In the case of BeeBox-CP, we performed additional static analysis on the BPF programs to verify that the context pointer is not leaked into other registers or spilled to memory. Due to calling convention considerations, we chose the callee-saved register `rbx` to hold the context pointer. We also duplicated the helper functions to avoid using or clobbering `rbx`, similar to `r12`.

XDP. Our support of XDP BPF programs includes two optimizations: BeeBox-RC (§5.4.1) and BeeBox-RB (§5.4.2). For XDP, the context is generated dynamically on the (kernel) stack; thus, we simply generate it on the BPF stack instead. In BeeBox-RC, part of the packet is copied into the `beebox` region. Although the copying only requires part of the packet, we allocate enough memory to fit potential XDP-supported modifications to the packet (e.g., extending the header). The range of the packet copied out is also adjusted in the event that packet pointers are modified in the XDP context. When enhanced with BeeBox-RB, depending on whether the network driver supports XDP natively, the allocation of the packet buffer at the driver level or the NAPI level is adjusted to appropriately handle allocating directly in the `beebox` region.

6.4 Engineering Effort

The core implementation of BeeBox required ≈900 additional lines of code (LOC) added to Linux v6.1. Specifically, we added ≈700 LOC to the BPF JIT engine and verifier to implement the main functionality and optimization schemes of BeeBox, and ≈200 LOC to adapt the hook points that invoke BPF programs. Atop this, we also migrated ≈1000 lines of helper code into separate compilation units and manually added the instrumentation at the source code as necessary. Regarding the helpers that we instrumented (§6.2): support for the three array map variants took a graduate student (with knowledge of Linux kernel internals) one day to develop; support for the two hash map variants took three days to develop; and the program-type-specific helpers took four hours.

7 Evaluation

We evaluated BeeBox in terms of effectiveness and performance. For the former, we analyzed the impact of BeeBox on BPF's functionality compared against no-mitigations and existing mitigations in Linux v6.1. For the latter, we evaluated BeeBox's performance on synthetic eBPF benchmarks and real-world BPF applications, including Katran, `seccomp-BPF`, and cBPF packet filtering. Our experiments were run on a machine equipped with a 16-core, 3.7GHz Intel Xeon W-2145 CPU and 64GB RAM, running 64-bit Ubuntu 20.04 LTS.

7.1 Effectiveness of BeeBox

7.1.1 Security Analysis

When hardened with BeeBox, *four* different kinds of pointers are considered for the security of the system. The first kind is pointers into untrusted regions, like the `beebox` and the kernel stack, when used by SU-hardened helpers. They are speculatively-safe because they cannot target outside the untrusted region(s) and there is no sensitive data in those regions. The second kind is clean pointers; they are speculatively-safe because they can only target the same locations as concrete execution, which is assumed to be trusted. The third kind is pointers used in ST-hardened functions. These pointers are secured since ST-hardened functions use speculation barriers to ensure no attacker-controlled speculation occurs during their execution. The forth kind is developer exempted pointers, including ones handled with `array_index_nospec`, whose value can be influenced by speculation, but are explicitly permitted by developers. In summary, BeeBox comprehensively protects all memory accesses during BPF execution.

7.1.2 Security Evaluation

We created three synthetic exploits that simulate speculative execution attacks launched from BPF programs, which attempt to (speculatively) access a target location to leak sensitive data. We consider BeeBox able to stop a given attack if we can no longer detect a signal in the cache corresponding to accessing the sensitive data. To probe the cache for a signal, we created a kernel module that performs Flush+Reload [102].

Exploit #1: Spectre-PHT. For the first exploit, we crafted a BPF program that performs a dynamically indexed access into a BPF map element. Here, the mis-speculation starts from a conditional branch, similar to the original Spectre attack [56]. The exploit works without mitigations and fails against BeeBox (and LPM).

Exploit #2: Spectre-STL. For the second exploit, we targeted Spectre-STL. We crafted a BPF program that triggers erroneous store-to-load forwarding by saturating store ports [58]. Similar to the first exploit, this exploit works without mitigations and fails against BeeBox (and LPM).

```
1  u64 bpf_skb_load_helper_32(const struct sk_buff *skb,
2                             const void *data,
3                             int headlen,
4                             int offset) {
5      __be32 tmp, *ptr;
6      const int len = sizeof(tmp);
7      if (likely(offset >= 0)) {
8          if (headlen - offset >= len)
9              return get_unaligned_be32(data + offset);
10         ...
11     }
12     return -EFAULT;
13 }
```

Listing 3: The packet-loading helper.

Exploit #3: Spectre-PHT via Helper. The third exploit is similar to Exploit #1, but the vulnerability occurs in a BPF helper function. Listing 3 presents one such vulnerable helper that loads bytes from a packet. If `offset` is out-of-bounds, but `headlen` was not cached, then the conditional in Line 8 will be speculated and the out-of-bounds address will be loaded in Line 9. For simplicity, we edited the kernel to flush the memory of `headlen` before calling the BPF program to force speculation. The BPF program we ran is a `socket` BPF program that triggers this mis-speculation and loads the out-of-bounds value. In the vanilla kernel, the attack triggers a memory load of the target location with or without LPM. In contrast, BeeBox prevents this exploit as it hardens this helper (SFI in BeeBox-RC and `array_index_nospec` in BeeBox-CP). This attack demonstrates that LPM offers partial coverage. Similar attacks can take place through packet filtering cBPF.

7.1.3 Compatibility Evaluation

The LPM prevent certain BPF program patterns from being accepted by the verifier. In contrast, BeeBox is designed with compatibility in mind, allowing more programs to pass verification, while also receiving superior protection. We describe two such examples, which we also summarize in Table 1.

Verifier State Explosion. The BPF verifier keeps an internal count of analyzed instructions to decide whether a program takes too long to verify and should be rejected—interestingly, an instruction may be analyzed multiple times if it is reachable from different paths. LPM can dramatically increase verification time since it requires all combinations of branch choices to be analyzed, even if they are not possible. To demonstrate this problem, we crafted a BPF program that contains 23 conditional jumps out of 47 instructions in total. Loading this BPF program as an unprivileged user (i.e., the current Spectre-PHT mitigations are enabled) fails as it breaks the verifier's limit of one million analyzed instructions. Without the analysis for speculative branches, the verifier decides the BPF program is safe after processing 611 instructions. Since BeeBox comprehensively stops access of sensitive information from BPF programs, it avoids this analysis and achieves verification scalability as program size and complexity increase.

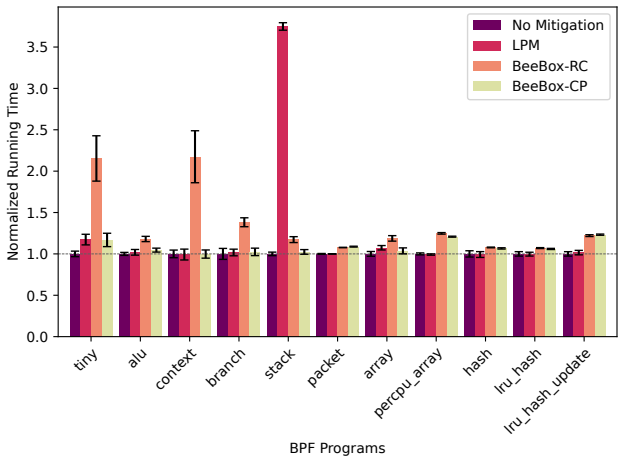


Figure 4: Synthetic micro-benchmark that involves various socket filter eBPF programs and different hardening schemes (no protection vs. LPM vs. BeeBox).

Conditional Pointer Arithmetics. Under the current Spectre-PHT mitigation(s), the BPF verifier rejects BPF programs where a pointer can have different value sets due to different branches (with the exception of map value pointers that point to different elements). This is a common code pattern in BPF programs; for example, one of Katran's BPF programs cannot be loaded by an unprivileged user since it has pointers that target different packet fields based on the IP version (e.g., IPv4 vs. IPv6). BeeBox can avoid this restriction; even though `boxptrs` can speculatively perform arbitrary arithmetic, they cannot point to sensitive data (outside `beebox`).

7.2 Performance of BeeBox

7.2.1 Basic Operations

We crafted a synthetic micro-benchmark that consists of small BPF programs (which we make available along with our prototype of BeeBox) that stress different kinds of operations, in order to better understand BeeBox's behavior in comparison with LPM. For each case, the running time of the given BPF program was measured using the `bpf` system call with the option `BPF_PROG_TEST_RUN`, which is modified to use `rdtscp` for accurate timing. We ran each BPF program in a tight loop for one million iterations with interrupts disabled. The BPF program type we use is `socket`, and the synthetic packets are 64 bytes, except in the `packet` benchmark, where the packet size is 150 bytes. Finally, all benchmark programs explicitly avoid using the stack unnecessarily.

Results are shown in Figure 4. We compared four different settings: (1) no mitigation, (2) LPM, (3) BeeBox-RC (§5.4.1), and (4) BeeBox-CP (§5.4.2). Setting (1) executes BPF programs as `root`, while (2) – (4) execute BPF programs in an unprivileged manner. LPM incur a 270% runtime over-

head compared to no protection (*No Mitigation*) on the `stack` benchmark, but do not exhibit any significant slowdown in any other case. Both BeeBox schemes incur a 7%–23% overhead on hash map operations due to the instrumentation of the helpers. BeeBox-RC has additional overhead due to copying, which peaks at 110% for the shortest BPF programs (`tiny` and `context`). This experiment also demonstrates that since LPM only add instrumentation around stack accesses, BeeBox's runtime performance gain over the LPM comes from better handling of stack load/store operations and Spectre-STL.

7.2.2 Real-world BPF Programs

Katran Benchmarks. Real-world eBPF programs can be more complex than the aforementioned, synthetic micro-benchmark programs, unavoidably using the stack for register spills. The performance of the LPM severely degrades in such situations, whereas BeeBox scales better with the complexity of the BPF program. To test this, we evaluated BeeBox on Katran's load balancer XDP program. In this setting, we benchmarked two BeeBox optimizations, applied separately to XDP: BeeBox-RC (§5.4.1) and BeeBox-RB (§5.4.2). The benchmarked BPF program is the standalone Katran XDP program, and the driving workload comes from Katran's `-perf_testing` option in its tester program. To run Katran's XDP program as a non-`root` user we provided the corresponding process with `CAP_NET_ADMIN` and `CAP_BPF`.

The results of the benchmark are shown in Figure 5. The overhead of LPM is consistently above 70%, with an average overhead of 112%, which confirms our hypothesis that most of the overhead is caused by register spilling. The overhead of BeeBox-RC ranges between 28%–50% in most benchmarks, but spikes up to 200% for packets that are passed-through or dropped early; therefore, on average, it has an overhead of 71%. For BeeBox-RB, the average overhead is 20%, with a maximum overhead of 39% when the workload consists of IPv6 ICMP packets (workload 9). When we compare BeeBox-RB to BeeBox-RC, we observe that the copying of context contributes 42% overhead on average. We also constructed a modified version of BeeBox-RB that applies instrumentation only to helper functions to determine their contribution to the overall overhead. This modified scheme has an average overhead of ≈17%, indicating that most of the overhead in BeeBox-RB is due to the instrumentation of the helpers.

Packet Filter Benchmark. We tested the performance of raw socket filtering using cBPF. To efficiently support cBPF packet filtering, we applied the BeeBox-CP scheme (§5.4.2). On the benchmark machine, we setup a UDP server listening on `localhost` and a UDP client that continuously sends packets with 32-byte payloads. We then attached a cBPF filter to a raw socket of the loopback (`lo`) device; the filter on the raw socket gets executed in the softIRQ raised by the client sending the packets, and is thus on the same CPU as the client. We pinned the client process to a CPU, and made sure that

Filter	No Mitigation	BeeBox-CP	%-Chg
`bpf1`	325927 (± 3611)	327778 (± 3006)	+0.57%
`bpf2`	324615 (± 3960)	323374 (± 5375)	−0.38%
`bpf3`	324114 (± 3977)	323834 (± 5088)	−0.09%
`bpf4`	328610 (± 4827)	325568 (± 7818)	−0.93%
`bpf5`	328072 (± 3883)	325395 (± 7352)	−0.82%
`bpf6`	314801 (± 2025)	313618 (± 2650)	−0.38%

(a) Packet filtering performance in pkts/s with 95% CIs.

Benchmark	No Mitigation	BeeBox
Nginx	0.81% (± 1.09%)	0.32% (± 1.47%)
Redis	0.98% (± 0.44%)	0.84% (± 0.74%)

(b) Tput degradation of `seccomp-BPF` with 95% CIs.

Table 2: cBPF performance results.

the CPU's utilization was close to 100%. Finally, to collect the results, we measured this scenario's throughput (pkts/s) as packets are filtered using rules that are similar to prior works [50, 101]. Results are shown in Figure 2a; the overhead introduced by BeeBox is $< 1\%$.

`seccomp-BPF` Benchmark. Because `seccomp-BPF`'s context is generated on the stack, BeeBox supports it by replacing the (kernel) stack allocation with a BPF stack allocation. To measure BeeBox's impact on `seccomp-BPF` performance, we applied sysfilter [23], a framework that generates and enforces `seccomp-BPF` policies, to Nginx [3] and Redis [4], and measured the performance of the hardened binaries. Nginx was configured to have two working processes. The test traffic was generated by `wrk` [30] with two running threads, each opening 128 connections, requesting 1KB payloads in every request. Redis was benchmarked using `memtier` [86] with two worker threads, each having 128 clients. The `GET:SET` request ratio was 10:1, and the data object size was 32 bytes. Each experiment ran for one minute. We ensured the CPU running the Nginx server and the Redis server was saturated with the testing configuration. Given that we tuned the packet sizes to be small, both tasks are syscall-intensive. The Nginx experiment handles over 43.5K req/s, and the Redis experiment handles more than 155K req/s. The results are shown in Figure 2b. The JITed cBPF programs already have limited impact on the services, and hence our defenses do not incur any observable overhead.

Memory Usage. We ran the real-world benchmarks and monitored the system's total memory usage. We measured this in a KVM VM with 4 vCPUs and 8GB of RAM, hosted on the benchmarking machine. The results of this experiment are shown in Table 3. The memory usage of the Linux kernel with no mitigations is reported in the second column (*Vanilla Usage*). The memory usage for BeeBox is given in the third

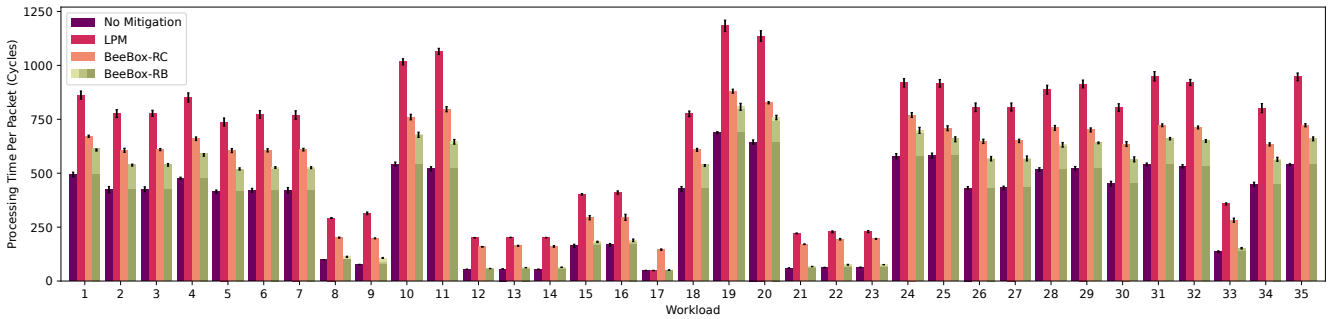


Figure 5: Katran's load balancer XDP program/benchmark. The most efficient scheme, BeeBox-RB, also includes a breakdown into *baseline*, *helper instrumentation*, and *BPF instrumentation* (colored dark to light).

Experiment	Vanilla Usage	BeeBox Usage	Overhead
At rest	176MB (178MB)	180MB (180MB)	2.4%
Packet filter	182MB (183MB)	186MB (188MB)	2.3%
Katran	580MB (582MB)	592MB (592MB)	2.0%
Nginx (`seccomp`)	189MB (190MB)	196MB (197MB)	3.5%
Redis (`seccomp`)	212MB (213MB)	218MB (221MB)	3.0%

Table 3: Memory usage of BeeBox compared to vanilla Linux. *At rest* means no workload is running. The reported numbers are formatted as '*avg (max)*'.

column. Overall BeeBox uses 4MB–6MB of additional memory when the application does not use BPF maps, and 12MB when it does, as is the case for Katran. Overall, the relative memory overhead ranges from 2%–3.5%.

8 Discussion

Compatibility. Regarding ISA-compatibility, although BeeBox is implemented for x86-64, the instrumentation techniques are generic and can be applied to other architectures. The main assumption required to enable our SFI scheme is that register-only masking and addition operations will enforce the pointer range during speculation. For existing BPF application compatibility, BeeBox does not require any changes from legacy BPF applications to benefit from its protection, unlike the current mitigations in the Linux kernel.

Future Extensions. To extend BeeBox to cover more BPF functionality, developer effort should be concentrated on migrating and instrumenting more helper functions and BPF map data structures. The division between kernel data and BPF data in BPF maps is ultimately developer-defined, therefore, there is no easy way to automate the entire process. However, `sparse` [64] has made the manual instrumentation process much easier and less error-prone for us. An extension to `sparse`'s capabilities to allow multiple pointer annotations working together (e.g., `__percpu` and `__beebox`) can streamline the process even more.

9 Related Work

Sandboxing BPF. SandBPF [62] also uses SFI [99] to sandbox BPF. However, it is designed to defend against errors in the verifier and the JIT compiler, and does not defend against transient execution attacks. It also assumes the input to the helpers is completely trusted. Lastly, its overhead is much higher than BeeBox: ≈7% in end-to-end experiments.

BPF Hardening. The BPF infrastructure has been used to assist kernel exploitation. BPF JIT-spraying [73, 88] and EPF [50] use BPF to bypass `ret2usr` defenses [20,54]. These attacks and proposed defenses revolve around memory safety, and assume the presence of memory-safety vulnerabilities. In contrast, BeeBox deals with transient execution attacks, which do not strictly require the presence of memory errors.

Correctness of BPF. Given the complexity of the BPF infrastructure, techniques have been proposed to enhance its security. Nelson et al. [79] proposed to use formal verification to secure the JIT compiler, and Jia et al. [49] proposed to use Rust to improve the runtime security of helpers. Similar to the work regarding BPF hardening above, these works also focus on memory safety problems, and hence differ from BeeBox's focus on transient execution attacks.

Wasm Hardening. WebAssembly (Wasm) [34] is a portable, low-level bytecode designed, in part, to safely and efficiently run untrusted code in web browsers. Unfortunately, Wasm is susceptible to Spectre attacks, which allow the attacker to escape the sandbox and access sensitive data outside of it. To combat Spectre attacks in Wasm, Swivel [78] presents two hardening schemes: (1) a software-only SFI approach, similar to BeeBox; and (2) a hardware-assisted approach that makes use of Intel's CET and MPK hardware extensions [21]. While the SFI scheme of Swivel is similar in concept to BeeBox, the two schemes have separate, domain-specific requirements and features that mold their respective designs.

10 Conclusion

We presented the design, implementation, and evaluation of BeeBox: a new security architecture that hardens BPF against transient execution attacks, allowing the OS kernel to expose its functionality to unprivileged users and applications. BeeBox sandboxes the BPF runtime against speculative code execution in an SFI-like manner, and uses a combination of static analyses and domain-specific properties to selectively remove enforcement checks to improve performance. Our BeeBox prototype for the Linux kernel supports popular features (e.g., BPF maps and BPF helper functions) and incurs low runtime overhead against prevalent transient execution attacks, such as Spectre-PHT and Spectre-STL. On average, BeeBox incurs 20% overhead in Katran's macro-benchmarks and $< 1\%$ throughput degradation in end-to-end, real-world settings that involve `seccomp-BPF` and packet filtering.

Availability

Our prototype implementation of BeeBox is available at: https://gitlab.com/brown-ssl/beebox

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation (NSF), through award CNS-2238467. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government or NSF.

References

[1] `bpf-helpers(7)` – Linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[2] `libbpf`. https://github.com/libbpf/libbpf.

[3] Nginx. https://nginx.org.

[4] Redis. https://redis.io.

[5] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[6] Ashish Bijlani and Umakishore Ramachandran. Extension Framework for File Systems in User Space. In *USENIX Annual Technical Conference (ATC)*, pages 121–134, 2019.

[7] Daniel Borkmann. bpf: Fix pointer arithmetic mask tightening under state pruning. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e042aa532c84d18ff13291d00620502ce7a38dda.

[8] Daniel Borkmann. Re: direct packet access from `SOCKET_FILTER` program. https://lore.kernel.org/bpf/06628370-b776-74a6-cbc0-5421989c64eb@iogearbox.net/.

[9] Brendan Gregg. Linux Extended BPF (eBPF) Tracing Tools. https://www.brendangregg.com/ebpf.html.

[10] Bugtraq. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium (SEC)*, pages 249–266, 2019.

[12] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N Khasawneh. Evolution of Defenses against Transient-Execution Attacks. In *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pages 169–174, 2020.

[13] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium (SEC)*, pages 249–266, 2019.

[14] Canella, Claudio and Genkin, Daniel and Giner, Lukas and Gruss, Daniel and Lipp, Moritz and Minkin, Marina and Moghimi, Daniel and Piessens, Frank and Schwarz, Michael and Sunar, Berk and Van Bulck, Jo, and Yarom, Yuval. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM Conference on Computer and Communications Security (CCS)*, pages 769–784, 2019.

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

[16] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *USENIX Operating Systems Design and Implementation (OSDI)*, pages 147–160, 2006.

[17] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical Foundations for Software Spectre Defenses. In *IEEE Symposium on Security and Privacy (S&P)*, pages 666–680, 2022.

[18] Jonathan Corbet. BPF: the universal in-kernel virtual machine. https://lwn.net/Articles/599755/.

[19] Jonathan Corbet. Reconsidering unprivileged BPF. https://lwn.net/Articles/796328/.

[20] Jonathan Corbet. Supervisor mode access prevention. https://lwn.net/Articles/517475/.

[21] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2023.

[22] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[23] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 459–474, 2020.

[24] Victor Duta, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. PIBE: Practical Kernel Control-Flow Hardening with Profile-Guided Indirect Branch Elimination. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 743–757, 2021.

[25] Jake Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[26] Michael Franz. E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism. In *New Security Paradigms Workshop (NSPW)*, pages 7–16, 2010.

[27] Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[28] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2023.

[29] Luis Gerhorst. bpf: Fix pointer-leak due to insufficient speculative store bypass mitigation. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/commit/?id=e4f4db47794c9f474b184ee1418f42e6a07412b6.

[30] Will Glozer. `wrk` – a HTTP benchmarking tool. https://github.com/wg/wrk.

[31] Tcpdump Group. `tcpdump`. https://www.tcpdump.org, 2023.

[32] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems (ESSoS)*, pages 161–176, 2017.

[33] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium (SEC)*, pages 897–912, 2015.

[34] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 185–200, 2017.

[35] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier Science, 2017.

[36] 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 *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 54–66, 2018.

[37] Guangyuan Hu, Zecheng He, and Ruby B. Lee. SoK: Hardware Defenses Against Speculative Execution Attacks. In *International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 108–120, 2021.

[38] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 969–986, 2016.

[39] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis.

ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 588–604, 2021.

[40] Intel. Analysis of Speculative Execution Side Channels. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/analysis-speculative-execution-side-channels.html.

[41] Intel. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

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

[43] Intel. Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html.

[44] Intel. Single Thread Indirect Branch Predictors. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html.

[45] Intel. Software Security Guidance: Advisory Guidance. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/advisory-guidance.html.

[46] IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[47] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1868–1882, 2018.

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

[49] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V Le, and Tianyin Xu. Kernel extension verification is untenable. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 150–157, 2023.

[50] Di Jin, Vaggelis Atlidakis, and Vasileios P. Kemerlis. EPF: Evil Packet Filter. In *USENIX Annual Technical Conference (ATC)*, pages 735–751, 2023.

[51] Jonathan Corbet. eBPF `seccomp()` filters. https://lwn.net/Articles/857228/.

[52] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 605–620, 2021.

[53] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium (SEC)*, pages 957–972, 2014.

[54] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *USENIX Security Symposium (SEC)*, pages 459–474, 2012.

[55] Ofek Kirzner and Adam Morrison. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *USENIX Security Symposium (SEC)*, pages 2399–2416, 2021.

[56] 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. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[57] Piotr Krysiuk and Benedict Schlueter. bpf: Fix leakage due to insufficient speculative store bypass mitigation. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2039f26f3aca5b0e419b98f65dd36481337b86ee.

[58] Piotr Krysiuk, Benedict Schlüter, and Daniel Borkmann. BPF and Spectre: Mitigating transient execution attacks. https://popl22.sigplan.org/details/prisc-2022-papers/11/BPF-and-Spectre-Mitigating-transient-execution-attacks.

[59] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea nd R. Sekar, and Dawn Song. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, 2014.

[60] Per Larsen, Stefan Brunthaler, and Michael Franz. Security through Diversity: Are We There Yet? *IEEE Security & Privacy*, 12:28–35, 2014.

[61] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX Security Symposium (SEC)*, pages 177–194, 2019.

[62] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. In *Workshop on eBPF and Kernel Extensions (eBPF)*, pages 42–48, 2023.

[63] Linux Kernel. Kernel Key Retention Service. https://docs.kernel.org/security/keys/core.html.

[64] Linux Kernel. Linux Documentation on Sparse. https://sparse.docs.kernel.org/en/latest/.

[65] Linux Kernel. Linux Documentation on Speculation. https://www.kernel.org/doc/Documentation/speculation.txt.

[66] Linux Kernel. Page Table Isolation (PTI). https://www.kernel.org/doc/html/latest/arch/x86/pti.html.

[67] Linux Kernel. Perf events and tool security. https://docs.kernel.org/admin-guide/perf-security.html.

[68] Linux Kernel. Program Types and ELF Sections. https://docs.kernel.org/bpf/libbpf/program_types.html.

[69] Linux Kernel. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[70] Linux Kernel. Spectre Side Channels. https://docs.kernel.org/admin-guide/hw-vuln/spectre.html.

[71] Liu, Fangfei and Yarom, Yuval and Ge, Qian and Heiser, Gernot and Lee, Ruby B. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy (S&P)*, pages 605–622, 2015.

[72] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *ACM Conference on Computer and Communications Security (CCS)*, pages 2109–2122, 2018.

[73] Keegan McAllister. Attacking hardened Linux systems with kernel JIT spraying. https://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html.

[74] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter Conference*, 1993.

[75] Marius Momeu, Fabian Kilger, Christopher Roemheld, Simon Schnückel, Sergej Proskurin, Michalis Polychronakis, and Vasileios P. Kemerlis. ISLAB: Immutable Memory Management Metadata for Commodity Operating System Kernels. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*, 2024.

[76] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258, 2009.

[77] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *International Symposium on Memory Management (ISMM)*, pages 31–40, 2010.

[78] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security Symposium (SEC)*, pages 1433–1450, 2021.

[79] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 41–61, 2020.

[80] Nikita Shirokov and Ranjeeth Dasineni. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[81] Aleph One. Smashing The Stack For Fun And Profit. *Phrack Magazine*, 7(49), 1996.

[82] Marco Patrignani and Marco Guarnieri. Exorcising Spectres with Secure Compilers. In *ACM Conference on Computer and Communications Security (CCS)*, pages 445–461, 2021.

[83] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. kRˆ X: Comprehensive Kernel Protection

against Just-In-Time Code Reuse. In *European Conference on Computer Systems (EuroSys)*, pages 420–436, 2017.

[84] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*, pages 563–577, 2020.

[85] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security Symposium (SEC)*, pages 1451–1468, 2021.

[86] Redis Labs. `memtier_benchmark`. https://github.com/RedisLabs/memtier_benchmark.

[87] Release Notes for Debian 11 (bullseye), 64-bit PC. Linux disables unprivileged calls to `bpf()` by default. https://www.debian.org/releases/bullseye/amd64/release-notes/ch-information.en.html#linux-unprivileged-bpf.

[88] Elena Reshetova, Filippo Bonazzi, and N Asokan. Randomization Can't Stop BPF JIT Spray. In *International Conference on Network and System Security (NSS)*, pages 233–247, 2017.

[89] Marc Schink and Johannes Obermaier. Taking a Look into Execute-Only Memory. In *USENIX Workshop on Offensive Technologies (WOOT)*, August 2019.

[90] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM Conference on Computer and Communications Security (CCS)*, page 753–768, 2019.

[91] Yafang Shao. bpf: Fix issue in verifying `allow_ptr_leaks`. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d75e30dddf73449bc2d10bb8e2f1a2c446bc67a2.

[92] Yafang Shao. bpf: Fix issue in verifying `allow_ptr_leaks`. https://lore.kernel.org/bpf/20230913122514.89078-1-gerhorst@amazon.de/.

[93] Alexei Starovoitov. tracing, perf: Implement BPF programs attached to kprobes. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2541517c32be2531e0da59dfd7efc1ce844644f5.

[94] SUSE Support. Security Hardening: Use of eBPF by unprivileged users has been disabled by default. https://www.suse.com/support/kb/doc/?id=000020545.

[95] The Cilium Authors. Cillium: eBPF-based Networking, Observability, Security. https://cilium.io/.

[96] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Peter C Johnson, and Kevin RB Butler. LBM: A Security Framework for Peripherals within the Linux Kernel. In *IEEE Symposium on Security and Privacy (S&P)*, pages 967–984, 2019.

[97] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 86–106, 2012.

[98] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-Flight Data Load. In *IEEE Symposium on Security and Privacy (S&P)*, pages 88–105, 2019.

[99] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient Software-based Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[100] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated Discovery of Microarchitectural Side Channels. In *USENIX Security Symposium (SEC)*, pages 1–18, 2021.

[101] Zhenyu Wu, Mengjun Xie, and Haining Wang. Swift: A Fast Dynamic Packet Filter. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 279–292, 2008.

[102] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium (SEC)*, pages 719–732, 2014.

[103] Fenghua Yu. Enable SMEP CPU Feature. https://lore.kernel.org/lkml/1305581685-5144-1-git-send-email-fenghua.yu@intel.com/.

[104] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 375–393, 2022.

A Artifact Appendix

A.1 Abstract

This is the artifact appendix for BeeBox: a new security architecture that hardens BPF against transient execution attacks. This appendix contains instructions about how to setup, run, and reproduce the results of BeeBox, along with information regarding system and resource requirements.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact consists of scripts for setting up QEMU/KVM virtual machines (VMs) for reproducing the main experiments of BeeBox. The majority of operations on the host are unprivileged, except for a handful required to create a Debian Linux root file system that is shared across the VMs. These privileged operations are all contained in `syzkaller`'s script, `create-image.sh`. We recommend enabling password-less `sudo` while running the script to streamline its execution.

A.2.2 How to access

Our artifact is publicly available on GitLab.

- Repository: https://gitlab.com/brown-ssl/beebox-ae
- Stable commit: be43784928ba43f0

The BeeBox Linux kernel is also publicly available on GitLab.

- Repository: https://gitlab.com/brown-ssl/beebox-linux
- Stable commit: 29e4d7de943cb43c

A.2.3 Hardware dependencies

The current prototype of BeeBox requires a machine equipped with a 64-bit x86 processor and at least 32GB of storage space. Additionally, while not a hard requirement, we recommend the machine has at least 4 CPU cores and 8GB of RAM.

A.2.4 Software dependencies

The artifact infrastructure depends on a three main packages:

- QEMU (KVM accelerated): for virtualizing the testing and benchmarking environment.
- SSH: for controlling the running VMs.
- Python3: for various scripting tasks.

On Debian GNU/Linux, these packages can be installed with:

```
$ sudo apt-get install qemu-system-x86 \
    openssh-client python3
```

Python packages `numpy` and `pyparsing` are required for summarizing benchmark results; these can be installed by running the following in the root of the artifact repository:

```
$ pip install -r requirements.txt
```

To check that KVM acceleration is available, the `cpu-checker` package can be installed and the `kvm-ok` program should be executed, as follows:

```
$ sudo apt-get install cpu-checker
$ sudo kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
```

When building the kernel variants from source the following build dependencies need to be satisfied as well (on Debian GNU/Linux):

```
$ sudo apt-get install build-essential bc kmod \
    cpio flex libncurses5-dev libelf-dev \
    libssl-dev dwarves bison
```

All other software dependencies—e.g., the BeeBox Linux kernel, Katran, `memtier`, `sysfilter`, Nginx, and Redis—are either included in the artifact repository as (stable) Git submodules or installed during the setup phase of the root file system. In both cases, everything is handled automatically and no additional work is required.

A.2.5 Benchmarks

This artifact provides three synthetic exploits and five benchmarks, corresponding to those found in Section 7 of the paper. In particular:

- `exploit`: a set synthetic programs demonstrating the effectiveness of BeeBox (§7.1.2); source code at `bpf_test/defense_effectiveness`.
- `micro`: a set of microbenchmarks (§7.2.1); source code at `bpf_test/micro_benchmark`.
- `katran`: a real-world eBPF benchmark (§7.2.2); source code at `katran`.
- `filter`: a packet filtering benchmark (§7.2.2); source code at `bpf_test/cbpf_socket_benchmark`.
- `nginx`: a syscall filtering benchmark (§7.2.2); source code at `bpf_test/seccomp_benchmark/nginx_test`.
- `redis`: a syscall filtering benchmark (§7.2.2); source code at `bpf_test/seccomp_benchmark/redis_test`.

A.3 Set-up

To reproduce the paper's major claims, we recreated the bare-metal benchmarking environment used in the paper with virtualization (i.e., QEMU/KVM). At a high-level, we first build four custom Linux kernels, namely:

- `vanilla`: a stock Linux kernel with some Spectre-PHT defenses disabled to allow Katran to run. Note that experiments with the LPM also use this kernel, but with defenses turned on.
- `hardened`: an implementation of BeeBox-RC.
- `optimized`: an implementation of BeeBox-CP and BeeBox-RB for socket filtering and XDP, respectively.

- `synthetic`: an exploit showcase; modifies the kernel to simulate an attacker's side-channel capabilities.

Then, we create a single root file system (shared across all VMs) that contains all of the benchmarks and scripts required to evaluate BeeBox. Finally, we virtualize these components to get an environment to reproduce BeeBox's results.

A.3.1 Installation

Prebuilt. To simplify evaluating BeeBox, we provide a prebuilt benchmarking environment that can be used as is (i.e., no building and installing). To use it, simply download the respective archive from Zenodo (https://zenodo.org/records/12212612/files/prebuilt.tar.gz) and decompress it. For example:

```
$ wget https://zenodo.org/records/12212612/\
files/prebuilt.tar.gz
$ tar -xzf prebuilt.tar.gz
```

Note that the prebuilt environment contains the entirety of the artifact repository, so if this option is chosen, there is no need to clone the `beebox-ae` repository from GitLab.

From scratch. To build the three kernels from scratch, enter the root of the artifact repository and run:

```
$ ./scripts/build_kernels.sh
```

Then, to build the root file system from scratch, run:

```
$ ./scripts/builds_rootfs.sh
```

After running these two commands, a root file system image and four built kernels will be found in the `build` directory. The installation process requires roughly 20GB of disk space; while compilation times vary by machine, an estimate based on our machine (8-core, 3.7GHz Intel Xeon W-2145 CPU with 64GB of DDR4 RAM) is roughly 1–2 hours.

A.3.2 Basic test

After the installation is complete, a shell to a VM running a stock kernel (i.e., the `vanilla` kernel with no hardening) can be obtained via:

```
$ ./scripts/run.sh vanilla
```

Successfully entering the VM with this command will ensure that the environment is setup correctly. The `vanilla` kernel configuration can also be swapped out for alternate configurations—namely, `lpm`, `hardened`, and `optimized`—to perform a more thorough "basic" test. Changes to the VMs do not persist across invocations (QEMU is invoked with `-snapshot`), so feel free to poke around!

A.4 Evaluation workflow

We have automated the evaluation workflow via the script `scripts/run.sh` (see its "help" menu for a complete summary of options). While the experiment descriptions below (§A.4.2) detail how to use this script to run each experiment individually, all experiments can also be batched together and run automatically via:

```
$ ./scripts/run.sh everything
```

This should take roughly 15–20 minutes to complete, producing results in both standard output and the `results` directory. At the end, it will also run `script/summary.py` to pretty-print tables summarizing the results.

Please note that while we use QEMU/KVM to recreate the environment used in the paper, the benchmark numbers presented in Section 7 of the paper were collected on bare-metal. As a result, there might be slight discrepancies between the reproduced results and those in the paper; however, overall trends should remain consistent.

A.4.1 Major Claims

(C1): *BeeBox mitigates speculative execution attacks launched from BPF programs. This is demonstrated by the experiment (E1) described in Section 7.1.2, whose results are illustrated in Table 1. In particular, BeeBox mitigates Spectre-PHT in BPF code and helpers, as well as Spectre-STL in BPF code.*

(C2): *BeeBox is more performant than the LPM for stack load and store operations and Spectre-STL. This is shown by the microbenchmarks in the experiment (E2) described in Section 7.2.1 with results shown in Figure 4.*

(C3): *BeeBox is more performant than the LPM in real-world eBPF programs. This is shown by benchmarking Katran's load balancer in the experiment (E3) described in Section 7.2.2, whose results are shown in Figure 5.*

(C4): *BeeBox exhibits $<1\%$ throughput degradation in end-to-end, real-world settings that involve packet filtering and `seccomp-BPF`. This is shown by the experiments (E4 and E5) described in Section 7.2.2, whose results are illustrated in Table 2.*

A.4.2 Experiments

(E1): *[Exploit Mitigation] [5 human-minutes + ≈ 0 compute-hours + $<$ 1GB disk]: demonstrates the defense effectiveness of BeeBox by showing it stops three synthetic exploits making speculative, out-of-bound accesses.*

How to: This experiment demonstrates the effectiveness of BeeBox via three synthetic exploits that run on the `synthetic` and `optimized` kernel configurations against no defenses, the LPM, and BeeBox. The exploits rely on a custom kernel module that simulates an attacker making speculative out-of-bounds accesses. This

experiment corresponds to the description presented in Section 7.1.2. The source code for the test can be found in `bpf_test/defense_effectiveness`, which should already be copied in the root file system image.

Preparation: None.

Execution: First, run all exploits on an undefended kernel. To do this, enter the `synthetic` kernel with exploits initialized by running the following:

```
$ ./scripts/run.sh synthetic exploit
```

This will drop you into a shell. Verify that the exploit-helper kernel module, named `ctest`, is loaded:

```
(vm)$ lsmod
Module                  Size  Used by
ctest                  16384  0
```

Then, enter the `bpf_test/defense_effectiveness` directory and run the three exploits without any defenses:

```
(vm)$ sudo ./pht_exp
(vm)$ sudo ./stl_exp
(vm)$ sudo ./pht_helper_exp
```

Save the output of these three commands, and then run them again without `sudo` to enable the LPM defenses:

```
(vm)$ ./pht_exp
(vm)$ ./stl_exp
(vm)$ ./pht_helper_exp
```

Save the output of these three commands, and `poweroff` the VM. Next, run the exploits against a BeeBox-hardened kernel by booting into a BeeBox VM:

```
$ ./scripts/run.sh optimized exploit
```

As before, check that the kernel module is installed, and then navigate to the `bpf_test/defense_effectiveness` directory and run the exploits:

```
(vm)$ ./pht_exp
(vm)$ ./stl_exp
(vm)$ ./pht_helper_exp
```

Save the output of these three commands.

Result: When a synthetic exploit succeeds (i.e., defenses fail), it means that out-of-bounds memory is accessed speculatively, which is determined by timing the reload of the memory. The corresponding output should look like:

```
(vm)$ sudo ./pht_exp
[+] reload takes 64 cycles,
        in-cache reload takes 66 cycles
[+] Speculative out-of-bound access succees!
```

If a defense successfully blocks an exploit, the corresponding output should look like:

```
(vm)$ ./pht_exp
[+] reload takes 318 cycles,
        in-cache reload takes 56 cycles
[-] Speculative out-of-bound access fail!
```

An undefended kernel fails to block all three exploits; against LPM only `pht_helper_exp` succeeds; and against BeeBox all exploits are defeated. Note that the Spectre-PHT attacks have a high probability of success, while Spectre-STL attacks may need to run multiple times to succeed. To get more consistent results, try to run the experiments multiple times. For example:

```
(vm)$ for i in {1..100}; do sudo ./stl_exp; \
done | grep -q "succeed" && \
echo "succeed" || echo "fail"
```

(E2): *[Microbenchmarks] [*1 *human-minute* + 0.1 *compute-hour* + < 1*GB disk]: run a suite of microbenchmarks across four kernel configurations. Expect the LPM overhead for the* `stack` *benchmark to be more than* 250%.

How to: This experiment runs a microbenchmark across the vanilla, LPM, BeeBox-RC, and BeeBox-CP kernel configurations, corresponding to the description in Section 7.2.1 and results presented in Figure 4. The source code for the benchmarks can be found in `bpf_test/micro_benchmark`, which should already be copied into the root file system image.

Preparation: Enter the repository root and ensure that the host machine is sufficiently quieted.

Execution: Select the `micro` test option of the `run.sh` script for each kernel configuration to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla micro
$ ./scripts/run.sh lpm micro
$ ./scripts/run.sh hardened micro
$ ./scripts/run.sh optimized micro
```

Each command will print raw results to standard output.

Results: To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py micro
```

The results should show that LPM for the `stack` benchmark has significant overhead (> 250% in our testing), much higher than the other schemes, while BeeBox's overhead is higher in other benchmarks, but of smaller magnitude. Further, `optimized` kernel configuration (BeeBox-CP) should have less (average) overhead when compared to the `hardened` configuration (BeeBox-RC).

(E3): *[Katran Benchmark] [*1 *human-minute* + 0.1 *compute-hour* + < 1*GB disk]: benchmark Katran's load balancer across four kernel configurations. Expect the LPM configuration to exhibit higher than* 100% *overhead, the hardened configuration around* 50–80% *overhead, and the optimized configuration exhibits* 15–30% *overhead.*

How to: This experiment benchmarks Katran's load balancer XDP eBPF program across the vanilla, LPM, BeeBox-RC, and BeeBox-RB kernel configurations, corresponding to the description in Section 7.2.2 and results presented in Figure 5. The source code for the benchmarks can be found in `katran`, which should already be copied into the root file system image and built.

Preparation: Enter the repository root and ensure that the host machine is sufficiently quieted.
Execution: Select the `katran` test option of the `run.sh` script for each kernel configuration to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla katran
$ ./scripts/run.sh lpm katran
$ ./scripts/run.sh hardened katran
$ ./scripts/run.sh optimized katran
```

Each command will print raw results to standard output.
Results: To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py katran
```

The results should show that the LPM configuration incurs around 100% overhead, the `hardened` configuration incurs (BeeBox-RC) around 50% overhead, and the `optimized` configuration (BeeBox-RB) exhibits less than 20% overhead.

(E4): *[Filter Benchmark] [1 human-minute + 0.2 compute-hour + < 1GB disk]: benchmark the performance of raw socket filtering using cBPF with the BeeBox-CP scheme. Expect to see overhead < 1%.*
How to: This experiment benchmarks the performance of raw socket filtering using cBPF for the BeeBox-CP scheme. It corresponds to the description in Section 7.2.2 and the results in Table 2a. The source code for this experiment can be found in `bpf_test/cbpf_socket_benchmark`, which should already be copied into the root file system.
Preparation: Enter the repository root and ensure that the host machine is sufficiently quieted.
Execution: Select the `filter` test option of the `run.sh` script for the `vanilla` and `optimized` kernel configurations to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla filter
$ ./scripts/run.sh optimized filter
```

Each command will print raw results to standard output.
Results: To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py filter
```

The results should show that the `optimized` version of BeeBox (BeeBox-CP) incurs < 1% overhead across the benchmark programs.

(E5): *[Seccomp-BPF Benchmark] [1 human-minutes + 0.3 compute-hours + < 1GB disk]: benchmark BeeBox's performance impact on `seccomp-BPF` for Nginx and Redis. Expect the throughput degradation of both applications to be < 1%.*
How to: This experiment benchmarks the performance of syscall filtering with `seccomp-BPF` across Nginx and Redis for the `optimized` BeeBox scheme. It corresponds to the description in Section 7.2.2 and the results in Table 2b. The source code for this experiment can be found in `bpf_test/seccomp_benchmark`, which should already be copied into the root file system.
Preparation: Enter the repository root and ensure that the host machine is sufficiently quieted.
Execution: Select the `nginx` and `redis` test options of the `run.sh` script for the `vanilla` and `optimized` kernel configurations to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla nginx
$ ./scripts/run.sh optimized nginx
$ ./scripts/run.sh vanilla redis
$ ./scripts/run.sh optimized redis
```

Each command will print raw results to standard output.
Results: To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py seccomp
```

The results should show that the `optimized` version of BeeBox incurs < 1% throughput degradation across the benchmark programs.

A.5 Notes on Reusability

To drop into a shell in one of the running kernels, the following command can be used:

```
$ ./scripts/run.sh [config] shell
```

where `config` is one of '`vanilla`', '`lpm`', '`hardened`', '`optimized`', or '`synthetic`'.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.