Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions

Elazar Gershuni
Tel Aviv University, Israel and
VMware Research, USA
elazarg@gmail.com

Nadav Amit
VMware Research, USA
namit@vmware.com

Arie Gurfinkel
University of Waterloo, Canada
arie.gurfinkel@uwaterloo.ca

Nina Narodytska
VMware Research, USA
nnarodytska@vmware.com

Jorge A. Navas
SRI International, USA
jorge.navas@sri.com

Noam Rinetzky
Tel Aviv University, Israel
maon@cs.tau.ac.il

Leonid Ryzhyk
VMware Research, USA
lryzhyk@vmware.com

Mooly Sagiv
Tel Aviv University, Israel
msagiv@cs.tau.ac.il

Abstract

Extended Berkeley Packet Filter (eBPF) is a Linux subsystem that allows safely executing untrusted user-defined extensions inside the kernel. It relies on static analysis to protect the kernel against buggy and malicious extensions. As the eBPF ecosystem evolves to support more complex and diverse extensions, the limitations of its current verifier, including high rate of false positives, poor scalability, and lack of support for loops, have become a major barrier for developers.

We design a static analyzer for eBPF within the framework of abstract interpretation. Our choice of abstraction is based on common patterns found in many eBPF programs. We observed that eBPF programs manipulate memory in a rather disciplined way which permits analyzing them successfully with a scalable mixture of very-precise abstraction of certain bounded regions with coarser abstractions of other parts of the memory. We use the Zone domain, a simple domain that tracks differences between pairs of registers and offsets, to achieve precise and scalable analysis. We demonstrate that this abstraction is as precise in practice as more costly abstract domains like Octagon and Polyhedra.

Furthermore, our evaluation, based on hundreds of real-world eBPF programs, shows that the new tool generates no more false alarms than the existing Linux verifier, while it supports a wider class of programs (including programs with loops) and has better asymptotic complexity.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314590

CCS Concepts • **Security and privacy → Operating systems security**; • **Software and its engineering → Automated static analysis**; *Software safety*.

Keywords ebpf, static analysis, linux, kernel extensions

ACM Reference Format:
Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages.
https://doi.org/10.1145/3314221.3314590

1 Introduction

We consider the problem of verifying *untrusted* kernel extensions. Modern operating systems achieve most of their functionality through dynamically loaded extensions that implement support for I/O devices, file systems, networking, etc. Extensions execute in the privileged CPU mode and must therefore be trusted by the system to contain no unsafe or malicious code. This trust is traditionally established through the use of testing to eliminate bugs and digital signing to prevent tampering. Additionally, formal verification tools are sometimes used to achieve stronger assurance [16, 33]. These tools are effective at finding bugs, but do not provide strong correctness guarantees.

Untrusted kernel extensions are a special type of extensions that originate from untrusted sources and therefore cannot be assumed safe even in the traditional best-effort sense. Such extensions can be installed by non-privileged users or contain untested code. Untrusted extensions allow applications to customize the kernel with application-specific packet processing [6] and security policies [10], install profiling, monitoring and debugging probes [8], and even modify how core kernel subsystems interact with the application [14].

In the past, operating systems relied on language-based techniques to sandbox untrusted extensions inside the kernel, including the use of domain-specific languages [17, 27] and bytecode interpreters [35]. This approach has become too restrictive and too expensive for many new use cases that require extensions with rich functionality and low overhead.

To mitigate these shortcomings, Linux recently adopted an alternative approach based on *automatic verification* [45]. The new technology, called *extended Berkeley Packet Filters* (eBPF), is built around a simple bytecode that gets compiled to native CPU instructions when the extension is loaded in the kernel. Unlike traditional bytecodes such as the Java bytecode, eBPF's compiler and runtime do not enforce type or memory safety. Instead, safety is enforced by a static verifier that checks that the program cannot access arbitrary kernel data structures or cause page faults (we give a more complete definition of safety in Section 4).

The Linux eBPF verifier implements an algorithm that, in a nutshell, tracks program state using bitmasks, smallest and largest possible values, and equivalence classes of values using identity-tracking. The content of the stack is tracked in certain cases. It enumerates all program paths while heuristically avoiding suffixes whose safety is implied by previously-seen paths. While this approach is adequate for simpler eBPF programs (with few instructions, mostly straight-line loop-free code with no complex pointer arithmetic), it is proving a major roadblock as new and more complex use cases are introduced [3, 6, 8, 11, 14].

There are four main issues eBPF developers are struggling with. First, the verifier reports many false positives, forcing developers to heavily massage their code for the verifier to accept it, e.g., by inserting redundant checks and redundant accesses (see Section 3). Second, the verifier does not scale to programs with a large number of paths (Section 3). Third, it does not currently support programs with loops. Finally, the verifier lacks a formal foundation. Its algorithm is not formally specified, and no formal argument about its correctness is given. This is a serious concern since a bug in the verifier causing it to accept unsafe programs introduces an easily exploitable kernel vulnerability. In fact, multiple such bugs have been discovered recently [31].

Two aspects of the eBPF verification problem appeal to the formal methods community. First, *the need for a better verifier is widely recognized by eBPF developers*. This is in contrast with many verification use cases, where verification tools are facing an adoption barrier, as developers are often reluctant to integrate these tools in their workflow, fix bugs reported by the tool, deal with false positives, etc. The eBPF community, on the other hand, has already embraced a development process where every program must pass the verifier. A better verifier, grounded in state-of-the-art verification theory and practice, would enable a wider range of eBPF use cases and dramatically simplify the development process.

Second, Linux executes untrusted extensions in a highly constrained environment. Conveniently, many of these constraints make verification tractable in practice. In particular, eBPF programs cannot perform dynamic allocation, access kernel data structures or call in-kernel APIs. They run in a single-threaded mode and their execution time is bounded to few thousand instructions. In addition, eBPF programs do not have indirect jumps; every jump instruction in the program points to a fixed location in the code.

Despite these restrictions, verifying eBPF programs is far from trivial due to their low-level nature, heavy use of pointers and pointer arithmetic, and reliance on register spilling. In this paper, we set out to develop an efficient verification algorithm for real-world eBPF programs that would overcome the limitations of the current verifier. The algorithm must support existing and emerging eBPF use cases and be provably sound. The algorithm does not have to be complete, i.e., it may fail to validate the safety of a well-behaved program. However, it should empirically report few false positives on a wide range of real-world programs.

We tackle the problem using the framework of *Abstract Interpretation* [22]. We seek an abstract domain and associated abstract transformers that can capture the relevant state of an eBPF program, while being efficient.

eBPF programs manipulate a fixed number of exclusively-owned memory regions. We use a numerical abstract domain to track pointer values by representing them as (*region*, *offset*) pairs. We observed that memory is manipulated in a disciplined way which permits tracking the *memory layout* of the bounded stack, while ignoring safe accesses to other regions, including the packet (whose size is not known in advance). Our experiments indicate that this abstraction does not lead to false positives.

We then turn to pick a numerical domain. Our goal is to determine the coarsest and most efficient abstraction that is precise enough for eBPF programs. We experiment with different abstractions, ranging from simple and lightweight ones like the Interval domain [21] to expensive relational domains like Octagon [38, 47] and Polyhedra [24]. We use a collection of 111 real-world benchmarks in this initial evaluation (see Section 7). Our results indicate that simple numerical domains such as Interval are not sufficiently expressive, while more powerful domains such as Octagon and Polyhedra are prohibitively expensive in the context of eBPF verification.

We strike the balance between speed and precision by choosing the Zone abstract domain [36]. Zone is a relational domain that supports predicates of the form $X - Y \leq C$, where X and Y are variables and C is a constant. It has cubic worst case complexity in theory and is fast in practice.

We implemented the proposed abstract domain in the Crab abstract interpretation framework [30]. In addition to the 111 benchmarks used in development, we applied the tool to 81 additional programs, some of which are relatively large and complex, with a single false positive. Our results can be summarized as follows:

Precision We show that our memory abstraction with Zone is sufficiently powerful to verify real-world eBPF programs. In particular, it is as precise in practice as the more costly abstractions such as Octagon and Polyhedra. It is furthermore at least as precise as the current Linux verifier, while being able to correctly verify programs where the Linux verifier returns false positives.

Performance We were able to verify each of the benchmark programs in 5.2 seconds or less, at average rate of roughly 1500 instructions per second. While the Linux verifier was faster on these benchmarks (most of which were hand-crafted to work well with the Linux verifier), our algorithm enjoys better asymptotic behavior, being able to verify programs where the Linux verifier times out.

Support for loops We, for the first time, enable verification of safety of eBPF programs with loops. To validate this capability, we created additional benchmarks with loops and successfully applied our tool to prove their safety. This is one of the most desired features by the Linux community, especially important to enable writing library functions.

2 Background on eBPF

eBPF bytecode is an evolution of the Berkeley Packet Filter (BPF) [35] technology that enabled safe execution of user-defined network packet filters inside the kernel. eBPF adapts the BPF instruction set to modern CPU architectures and a wider range of use cases [45]. It also introduces a richer memory model, described below. Most importantly, while BPF relied on an interpreter to safely execute programs, eBPF introduced a new toolchain consisting of a static verifier and compiler to the native CPU instruction set. eBPF bytecode can also run in an interpreter; however the interpreter assumes that the bytecode has passed the verifier and thus avoids costly runtime safety checks.

eBPF programs can be written directly in bytecode, but are typically written in C and compiled to bytecode by the `llvm` eBPF backend [7]. The `llvm` compiler is not part of the trusted computing base (TCB); therefore verification is performed at the bytecode level. eBPF instructions operate over eleven 64-bit registers `r0..r10`; Table 1 illustrates the four main classes of instructions.

eBPF programs can be attached to a predefined set of kernel events such as arrival of a packet at a network interface or execution of a system call by a process. Linux version 1.19 defines 21 event types and corresponding eBPF *program types*, and the list is growing rapidly.

Control flow. An eBPF program is triggered by an occurrence of the event the program is attached to. Execution starts at the first instruction and terminates at the `exit` instruction.

Table 1. Example of eBPF instructions.

Category	Example	Description
Arith.	`r1 += r2`	Add register `r2` to `r1`
Memory	`r1=*(u64*)(r2+3)`	Load 64-bit at address `r2+3`
Branch	`if r1<=r2 goto +5`	Skip 5 instructions if `r1<=r2`
Call	`call 5`	Call helper function #5

The verifier guarantees termination by disallowing programs with loops. Since all jump instructions have constant jump offsets, loop termination is trivially enforced by checking that the program control-flow graph (CFG) is acyclic. In this work, we relax this requirement and introduce support for verifying safety of programs with loops.

eBPF programs have two ways of calling external code. First, the Linux kernel exports a fixed set of *helper functions* that can be invoked at any point in the program. Helper functions have well-defined effect on registers, and each helper has a signature defining which memory locations it can access or modify. Second, a program can perform a tail-recursive call to another eBPF program. To guarantee termination, recursion depth has a constant limit at runtime. Tail recursion does not affect verification, since the safety of each program can be established in isolation.

Memory model. An eBPF program can access a fixed set of memory regions, known at compile time: (1) the `context` region stores fixed-size invocation arguments specific to the given program type; (2) the `packet` region stores variable-size arguments, e.g., a network packet; (3) the `stack` region is a 512-byte scratch memory, typically used as program stack. The program can acquire access to additional regions via the *maps* API [5]. Such regions can be shared by multiple processes, as well as between kernel and user-space applications. We discuss shared regions in Section 4.

All regions except `packet` have statically known sizes. The size of `packet` is established at runtime by reading its start and end addresses from predefined location inside the `context` region (see Section 3).

Verification goal. The eBPF verifier must establish the following three properties:

- *Memory safety:* The program is only allowed to access memory locations within its allocated regions.
- *Information flow security:* Many eBPF programs run on behalf of non-privileged users and are therefore not allowed to leak any internal kernel data structures (except the ones explicitly passed as arguments to the program) to the user. Memory safety ensures that the program cannot access any kernel state outside of its memory regions. The only remaining way for the program to observe secret kernel state is by reading uninitialized registers or stack locations. Therefore such uninitialized reads are considered safety violations.
- *Termination:* All program executions must terminate.

Table 2. Simple eBPF program. `data` and `data_end` variables point to the start and end of the `packet` region.

C code	Bytecode	Invariant
Precondition:	`data = *(u32*)(r1+76), data_end = *(u32*)(r1+80)`	
`long *start=(void*)ctx->data;`	`1: r3 = *(u32 *)(r1 + 80)`	`r3=data_end`
`long *end=(void*)ctx->data_end;`	`2: r1 = *(u32 *)(r1 + 76)`	`r3=data_end, r1=data`
`if (start+1 > end) return;`	`3: r2 = r1+8`	`r3=data_end, r1=data, r2=data+8`
	`4: if r2>r3 goto <EXIT>`	`r1=data, data+8 <= data_end`
	`assert r1 >= data && r1<=data_end-8`	
`*start = 0;`	`5: *(u64*)(r1) = 0`	
	`EXIT: exit`	

In this work, we focus on the first two properties. Our verifier does not currently implement termination check. All existing eBPF programs are acyclic and therefore trivially terminating. For programs with loops, our algorithm verifies safety, but not termination. See Section 6 for more details.

3 Motivating Examples

We motivate the design of our abstract domain by exploring common patterns found in real-world eBPF programs. We consider several example programs that summarize insights distilled from hundreds of real-world kernel extensions.

Example 3.1 (A simple eBPF program)**.** The program in Table 2 shows a common pattern found in many eBPF programs. The first column shows the C code for this example. The `ctx` variable is a pointer to the `context` region, whose content is a C struct that stores pointers to the start and the end of the `packet` region in `ctx->data` and `ctx->data_end` fields. The program checks if the `packet` region has enough space for an 8-byte write before performing the write.

The eBPF verifier operates on the bytecode representation of this program, shown in the second column. Before executing the program, the eBPF loader sets register `r1` to point to the start of the `context` region. The precondition in the top row of the table specifies the location of `packet` region pointers within `context` (here `data` and `data_end` are ghost variables pointing to the start and end of the region). The program reads these pointers in lines 1 and 2 and checks that the end address is at least 8 bytes larger than the start (lines 3 and 4). If so, it writes an 8-byte value at the start of the region (line 5). The assertion before line 5 is the safety condition, which states that the memory access falls within the bounds of the `packet` region. The last column of the table lists postconditions of each instruction sufficient to validate the assertion (in particular, the last postcondition `r1 = data, data+8 <= data_end` implies the assertion).

Note that even in this trivial program proving safety requires establishing invariants relating two program variables, e.g., `r2 = data+8`. We avoid this constraint using an offset-based encoding that models pointers as (region, offset) pairs, where the first component identifies the memory region the pointer addresses and the second component is the offset within the region (Section 4). Using this encoding, our tool generates the constraint `r2 = 8`.

Example 3.2 (Ternary invariant)**.** The program in Table 3 is similar to the first example, but uses a value read from `r5` as a variable offset into the `packet` region. Proving its safety requires ternary constraints, e.g., `data+r5+8 <= data_end`. The offset-based encoding only reduces this constraint to two variables, `r5+8 <= data_end`. This indicates that non-relational abstract domains, such as the Interval domain [21], are insufficient in eBPF verification.

Observation 1. *The analysis must track binary relations among registers.*

Table 3. A program that performs write at a variable offset and requires invariants over three variables.

Bytecode	Invariant
`1: r5 = ...`	(`r5` is initialized.)
`2: r3 = *(u32 *)(r1 + 80)`	`r3 = data_end`
`3: r1 = *(u32 *)(r1 + 76)`	`r3 = data_end, r1 = data`
`4: r2 = r1+r5`	`r3 = data_end, r1 = data, r2 = data+r5`
`5: if r2<r1 goto <EXIT>`	`r3 = data_end, r1 = data, r2 = data+r5, data+r5 >= data`
`6: r2 = r2+8`	`r3 = data_end, r1 = data, r2 = data+r5+8, data+r5 >= data`
`7: if r2>r3 goto <EXIT>`	`r1 = data, data+r5 >= data, data+r5+8 <= data_end`
`8: r1 = r1 + r5`	`r1 = data+r5, data+r5 >= data, data+r5+8 <= data_end`
`assert r1 >= data && r1<=data_end-8`	
`9: *(u64*)(r1) = 0`	

Table 4. Register spilling. This program is similar to the one in Table 2, but it additionally spills register `r3` on the stack in line 3 (eBPF register `r10` is an immutable pointer to the bottom of the `stack` region). The spilled value is loaded to register `r4` in line 4. Highlighted invariants show how information is tracked through the stack.

Bytecode	Invariant
`1: r3 = *(u32 *)(r1 + 80)`	`r3 = data_end`
`2: r1 = *(u32 *)(r1 + 76)`	`r3 = data_end, r1 = data`
`3: *(u64*)(r10-8) = r3`	`r3 = data_end, r1 = data,` **`*(u64*)(r10-8) = r3`**
`...`	(`r3` is overwritten, **`*(u64*)(r10-8) = data_end`**)
`4: r4 = *(u64*)(r10-8)`	**`r4 = data_end,`** `r1 = data`
`5: r2 = r1+8`	`r4 = data_end, r1 = data, r2 = data+8`
`6: `**`if`**` r2>r4 `**`goto`**` <EXIT>`	`r1 = data, data+8 <= data_end`
`assert`` r1 >= data && r1<=data_end-8`	
`7: *(u64*)(r1) = 0`	

All invariants we have encountered so far (with the exception of program preconditions) were over program registers. It is tempting to restrict our abstract domain to only such predicates, while abstracting away the content of memory. Although appealing from the performance perspective, such an abstraction would be imprecise in practice. When the working set of a program does not fit in registers, parts of it must be temporarily spilled to the stack.

Example 3.3 (Register spilling)**.** Table 4 shows a modified version of Example 3.1 that temporarily stores the value of `r3` on the stack. Proving safety of this code requires tracking memory content via the invariant `*(u64)(r10-8)=data_end` .

Observation 2. *The analysis must track values in memory, including relations between different locations, as if they were registers.*

Example 3.4 (Loops)**.** Consider the `strncmp` function in Figure 1. When `n` is known at compile time, the eBPF toolchain handles such code by inlining and unrolling the body of the function. This transformation is not applicable when `n` is variable, even if it has a known static bound, e,g.:

```
  if (n < 100) strncmp(s1, s2, n)
```

Furthermore, the `break` statements in the body of the loop lead to path explosion, e.g., consider the following program:

```
      strncmp(s1, s2, VALUE_SIZE);
      strncmp(s3, s4, VALUE_SIZE);
```

The program has the number of paths quadratic in `VALUE_SIZE`, which quickly overwhelms the path enumeration-based Linux eBPF verifier. (See Figure 11).

```
int strncmp(char* p1, char* p2, size_t n) {
    for (size_t i=0; i < n; i++) {
        if (p1[i] != p2[i]) return 0;
        if (p1[i] == '\0') break;
    }
    return 1;
}
```

Figure 1. eBPF program with a loop.

These issues severely limit the use of loops in eBPF programs.

Observation 3. *As eBPF programs are getting larger and more complex, verification via path enumeration is becoming infeasible. Abstract interpretation can potentially overcome the path explosion with the help of join and widening operators, which trade precision for performance.*

Summary. We briefly summarize the properties of eBPF programs that guide our choice of verification methodology. On the one hand, eBPF programs do not contain several sources of complexity common in software verification such as dynamic memory allocation, concurrency, and function pointers. In addition, none of the eBPF programs we have encountered manipulate complex data structures like lists, trees or maps. Finally, eBPF verification focuses on safety, as opposed to more complex properties like functional correctness or complex temporal properties.

On the other hand, the eBPF verifier must perform precise pointer analysis without relying on high-level type information, which is not available at the bytecode level. The analysis must be sound and produce few false positives. This requires tracking pointers and offsets through memory and registers. The analysis must handle programs with loops and should not explode with the number of program paths.

4 Programming Model

This section defines eBPFPL—a core low-level programming language for kernel extensions which captures the essence of eBPF programs. Section 4.1 provides the syntax of the language and Section 4.3 defines its concrete operational semantics. The semantics enforces safety at runtime by aborting into an error state when it detects a safety violation. The abstract interpretation algorithm in Section 5 conservatively over-approximates this semantics. Thus, if the analyzer manages to verify that a program never aborts, it effectively establishes that it is safe to execute the program in the kernel. The semantics abstracts away certain details regarding the treatment of maps, library functions and overflows (see

$$
\begin{array}{rcl}
cmd & ::= & w := E \mid w :=_{sz} *p \mid *p :=_{sz} x \\
& \mid & \mathtt{assume}(B) \mid w := \mathtt{shared}\, K \\
E & ::= & K \mid x \mid x+y \mid x-y \\
B & ::= & x = y \mid x \neq y \mid x \leq y
\end{array}
$$

Figure 2. Primitive commands. K denotes a numeral.

Section 6). In particular, in this section and in Section 5 we assume the semantics can represent numerical values using mathematical integers.

4.1 Core Programming Language for eBPF

As variables, eBPFPL programs use a fixed set of *registers* Register = $\{\mathtt{r0}, \ldots, \mathtt{r10}, \mathit{data\_start}, \mathit{data\_end}\}$, ranged over by meta-variables p, w, x, y.

An eBPFPL program is represented as a control graph whose edges are annotated with the primitive commands listed in Figure 2: A primitive command *cmd* is either an *assignment* of an expression E to a *register*, a byte addressable *load* or *store* of *sz* bytes, where *sz* is either one, two, four, or eight bytes, an `assume`(B) statement which filters out states in which the boolean condition B does not hold, or a `shared`K command which returns a pointer to a *shared* region of size K bytes. (We discuss shared regions below.)

4.2 Design Consideration

We motivate our formalization by discussing some of the peculiarities regarding the way eBPF programs access the memory, and our abstraction of these operations.

Memory regions. A memory region is a disjoint, contiguous and byte-addressable memory area. eBPF programs manipulate two kinds of regions: *private* regions, which can be accessed only by the program, and *shared* regions, which are used for intra-kernel inter-process communication.

Each eBPF program has three private regions: *context*, *stack*, and *packet*. The *context* region is a small read-only memory area of a compile-time known size and format which is used to transmit information from the kernel to the eBPF program. The *stack* region is comprised of 512 bytes which function as scratch memory which is mainly used for register spilling and transferring parameters to library functions. The *packet* region stores an incoming/outgoing network packet. The size of the packet is not known at compile time, and only an upper bound is known. Instead, pointers to the start and end of the packet are stored in predefined locations in the context. Our semantics checks that accesses to the private regions are within their bounds, however it only tracks the contents of the stack region: The packet region stores only numerical values which do not affect the safety of the program, and the only information our analysis needs from the context region is the size of the packet region. Thus, for simplicity, we assume to have two immutable registers pointing to the start (*data_start*) and the end (*data_end*) of the packet region.

Shared regions are used to share data between different running processes. As shared regions can be overwritten at any moment, our semantics does not keep track of their contents. Instead, it only verifies that they are not accessed out of bound. eBPFPL abstracts away the details of how shared regions are obtained. We use the `shared`K command which returns a pointer to the beginning of an arbitrary (fresh or existing) shared region of size K.

Values and tags. The values a program manipulates are either numbers or pointers. We record the values of pointers as offsets from the beginning of the region they point to. We distinguish numerical values from pointers using tags: A value tagged `num` is a numerical value, while a value tagged R is a pointer offset into region R.

Memory accesses. Memory regions are byte-addressable. For example, if p points to the beginning of the stack, then the command $*p :=_4 3$ writes the value 3 to the first four bytes in the stack. If the next command executed is $*p :=_2 13$ then the first two bytes in the stack are overwritten with the value 13, leaving the third and fourth byte with an implementation-dependent value.

Our analysis does not track partially-overwritten values: when the program loads an *indefinite* value, i.e., executes a load instruction that access bytes that were not the target of a single store operation (e.g., only loading the fourth byte after the store of 13), the result is a nondeterministically chosen value whose tag is either `num`, if all the loaded bytes contained numerical values, or the invalid tag `inv`, otherwise. We do so because we wish to allow unaligned, partial and overlapping accesses to numerical values, but not to pointers. This prevents gleaning information out of its byte-level representation, as could have happened if these bytes are treated as if they contain numerical value. Leaking such information is dangerous as it can allow malicious users to gain insight into the memory layout of the kernel. (Note that when an eBPF program executes on a standard machine such an accesses would return the actual contents of the memory.)

4.3 Concrete Semantics

We now present non-standard concrete semantics. The goal is to formalize the safety properties we validate, and to serve as a stepping stone towards the analysis by abstracting away certain details.

4.3.1 Machine States

Figure 3 defines the semantic domain of *machine states*. A machine state is a triple $\sigma = (e, \mu, \zeta)$ comprised of an *environment* e, which maps register names to their contents; a *memory* μ, which maps *memory cells*—subsegments of the stack region identified by their start address a and their size *sz*—to their contents; and ζ, a set of addresses in the stack that hold a number or part of it, but not a pointer, or parts of which. Registers and cells store *tagged values*, i.e., pairs (R, n) comprised of a tag $R \in \mathcal{R}$ and an integer $n \in \mathbb{Z}$. The set $\mathcal{R}$ contains the numerical tag (num), the invalid tag (inv), private region identifiers (ctx, stk, pkt), and shared region identifiers from the unbounded set Shared.

Notation. In the following, we denote the value and type of a register x in environment e by $e_n(x)$ and $e_\rho(x)$ respectively. Similarly, we denote the value and tag of every memory cell c in memory μ by $\mu_n(c)$ and $\mu_\rho(c)$, respectively.

Initial states. A state (e, μ, ζ) is an *initial state* if register `r10` points to the end of the stack, i.e., $e(\texttt{r10}) = (\mathsf{stk}, 512)$; $e(\mathit{data\_start}) = (\mathsf{pkt}, 0)$, and $e_\rho(\mathit{data\_end}) = \mathsf{pkt}$; register `r1` points to the beginning of context region, i.e., $e(\texttt{r1}) = (\mathsf{ctx}, 0)$; for any other register x, $e(x) = (\mathsf{inv}, 0)$; and no memory cell is present or might be considered to contain a numerical value, i.e., $\mathrm{dom}(\mu) = \emptyset$ and $\zeta = \emptyset$.

4.3.2 Operational Semantics

eBPFPL has a small-step operational semantics, which is an adaptation of a standard two-level store semantics to abort the program in a special error state $\lightning$ if it is about to perform an unsafe operation, and to treat loads and stores that overlap existing values in the aforementioned conservative way.

Formally, the semantics of a program is defined as a transition relation $\cdot \Rightarrow_{\lightning} \cdot$ which checks that executing the command is *safe* using the Safe() predicate before continuing according to the transition relation of safe commands $\Rightarrow$:

$$\langle cmd,\ \sigma\rangle \Rightarrow_{\lightning} \begin{cases} \sigma' & \mathsf{Safe}(cmd, \sigma) \wedge \langle cmd,\ \sigma\rangle \Rightarrow \sigma' \\ \lightning & \text{otherwise} \end{cases}$$

A state σ is *reachable* in a program P if there is an execution of P which starts at an initial state which produces σ. An eBPFPL program P is *safe* if does not reach the error state.

Enforcing safety. Executing a command is not safe if it results in a meaningless value (e.g., the sum of two pointers), leaks information regarding the layout of different regions (e.g., by comparing a pointer to any number other than zero), or leads to a memory fault (e.g., by writing outside a memory region). To enforce memory safety, we assume that when P executes it has access to an immutable *size map* $\mathit{sizeof} \in (\mathcal{R} \setminus \{\mathsf{num}, \mathsf{inv}\}) \to \mathbb{N}$ which gives the size of every memory region where $\mathit{sizeof}(\mathsf{stk}) = 512$.

We formalize the notion of safety using a predicate $\mathsf{Safe}(cmd, \sigma)$ which determines if it is safe to execute cmd on state $\sigma = (e, \mu, \zeta)$. The safety predicate is a conjunction of a generic condition Safe_{inv} and a command-specific condition Safe_{cmd}, i.e., $\mathsf{Safe}(cmd, \sigma) = \mathsf{Safe}_{inv}(cmd, \sigma) \wedge \mathsf{Safe}_{cmd}(\sigma)$.

The generic condition states that no register mentioned in a statement whose value is read can hold an invalid value:

$$\mathsf{Safe}_{inv}(cmd, \sigma) = \mathsf{inv} \notin \{e_\rho(x) \mid x \in \mathsf{ReadRegs}(cmd)\}\,,$$

where ReadRegs(cmd) denotes the set of registers whose values are read in cmd.

$$\begin{array}{rcrcl} & & \mathcal{R} & = & \mathsf{Shared} \cup \{\mathsf{ctx}, \mathsf{stk}, \mathsf{pkt}, \mathsf{num}, \mathsf{inv}\} \\ a & \in & \mathsf{Address} & = & \{0, \ldots, 511\} \\ e & \in & \mathsf{Env} & = & \mathsf{Register} \to (\mathcal{R} \times \mathbb{Z}) \\ c & \in & \mathsf{Cell} & = & \mathsf{Address} \times \mathsf{Size} \\ \mu & \in & \mathsf{Mem} & = & \mathsf{Cell} \hookrightarrow (\mathcal{R} \times \mathbb{Z}) \\ \zeta & \in & \mathsf{Format} & = & 2^{\mathsf{Address}} \\ \sigma & \in & \mathsf{State} & = & \mathsf{Env} \times \mathsf{Mem} \times \mathsf{Format} \end{array}$$

Figure 3. Semantic domains.

We now specify command-specific safety conditions.

An assignment $w := E$ is safe unless its evaluation leads to undefined pointer arithmetics operations, i.e., its evaluation either subtracts two pointers to the same region, or adds (subtracts) a numerical value to (from) a pointer. Specifically, pointers to distinct regions cannot be subtracted.

$$\mathsf{Safe}_{w\,:=\,E}(\sigma) = \begin{cases} e_\rho(x) = \mathsf{num} \vee e_\rho(y) = \mathsf{num} & E = x + y \\ e_\rho(x) = e_\rho(y) \vee e_\rho(y) = \mathsf{num} & E = x - y \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Filtering states using an `assume`(B) command is safe if it does not leak information regarding the relative addresses of different memory regions, e.g., by comparing a pointer to a non-zero number or testing the relative order of pointers to distinct regions. (Below, $\bowtie \in \{=, \neq\}$.)

$$\begin{array}{rcl} \mathsf{Safe}_{\mathtt{assume}(x \bowtie y)}(\sigma) & = & e_\rho(x) = e_\rho(y) \vee e(x) = (\mathsf{num}, 0) \\ & & \vee\ e(y) = (\mathsf{num}, 0) \\ \mathsf{Safe}_{\mathtt{assume}(x \le y)}(\sigma) & = & e_\rho(x) = e_\rho(y) \end{array}$$

Load and store commands are safe if they only access bytes within the region, and do not write pointers to externally-visible locations:

$$\begin{array}{rl} \mathsf{Safe}_{w\,:=_{sz}\,*p}(\sigma) = & \mathit{inbounds}(e_\rho(p), e_n(p), sz) \wedge e_\rho(p) \neq \mathsf{num} \\ \mathsf{Safe}_{*p\,:=_{sz}\,x}(\sigma) = & \mathit{inbounds}(e_\rho(p), e_n(p), sz) \wedge e_\rho(p) \neq \mathsf{num} \\ & \wedge\ e_\rho(x) \neq \mathsf{num} \to e_\rho(p) = \mathsf{stk} \end{array}$$

$$\mathit{inbounds}(R, a, sz) = \begin{cases} 0 \le a \le e_n(\mathit{data\_end}) - sz & R = \mathsf{pkt} \\ 0 \le a \le \mathit{sizeof}(R) - sz & \text{otherwise} \end{cases}$$

Note that the bound check for the packet region is done with respect to *data_end* and not *data_end* − *data_start*. This is because *data_start* points to the beginning of the region and thus its offset is zero.

Meaning of safe commands. Figure 4 defines the meaning of primitive commands whose execution is deemed to be safe. (We use $\overline{(a, sz)} = \{i \in \mathbb{Z} \mid a \le i < a + sz\}$ to denote set of integers from $a \in \mathbb{Z}$ to $a + sz - 1$, where $sz \in \mathsf{Size}$.)

The meaning of assignments is quite standard. Note that pointer arithmetics between a pointer to region R and a number results in a pointer to region R and that it is possible that the pointer's offset would be out of bounds, but any attempt to dereference such a pointer would abort the program.

$$\langle \texttt{w := K},\ \sigma\rangle \Rightarrow (e[w \mapsto (\mathsf{num}, K)], \mu, \zeta)$$
$$\langle \texttt{w := x},\ \sigma\rangle \Rightarrow (e[w \mapsto e(x)], \mu, \zeta)$$
$$\langle \texttt{w := x + y},\ \sigma\rangle \Rightarrow (e[w \mapsto (R, e_n(x) + e_n(y))],\ \mu, \zeta)$$
$$\text{where } R = \text{ if } (e_\rho(x) = \mathsf{num}) \text{ then } e_\rho(y) \text{ else } e_\rho(x)$$
$$\langle \texttt{w := x - y},\ \sigma\rangle \Rightarrow (e[w \mapsto (R, e_n(x) - e_n(y))],\ \mu, \zeta)$$
$$\text{where } R = \text{ if } (e_\rho(x) = e_\rho(y)) \text{ then num else } e_\rho(x)$$

$$\langle \texttt{w := shared}\,K,\ \sigma\rangle \Rightarrow (e[w \mapsto (R, 0)], \mu, \zeta)$$
$$\text{where } R = \mathsf{num} \vee (R \in \mathsf{Shared} \wedge \mathit{sizeof}(R) = K)$$

$$\langle \texttt{*p :=}_{sz}\, x,\ \sigma\rangle \Rightarrow (e, \mu', \zeta') \qquad \text{if } e_\rho(p) = \mathsf{stk}$$
$$\text{where } (\mu', \zeta') = \mathit{Store}(\mu, \zeta, (e_n(p), sz), e(x))$$
$$\langle \texttt{*p :=}_{sz}\, x,\ \sigma\rangle \Rightarrow \sigma \qquad \text{if } e_\rho(p) \neq \mathsf{stk}$$

$$\langle \texttt{w :=}_{sz}\ \texttt{*p},\ \sigma\rangle \Rightarrow (e[w \mapsto v], \mu, \zeta) \qquad \text{if } e_\rho(p) = \mathsf{stk}$$
$$\text{where } v \in \mathit{Load}(\mu, \zeta, (e_n(p), sz))$$
$$\langle \texttt{w :=}_{sz}\ \texttt{*p},\ \sigma\rangle \Rightarrow (e[w \mapsto v], \mu, \zeta) \qquad \text{if } e_\rho(p) \neq \mathsf{stk}$$
$$\text{where } v = (\mathsf{num}, \beta) \wedge \beta \in \mathbb{Z}$$

$$\langle \texttt{assume(x=y)},\ \sigma\rangle \Rightarrow \sigma \qquad \text{if } e_n(x) = e_n(y) \wedge e_\rho(x) = e_\rho(y)$$
$$\langle \texttt{assume(x}\neq\texttt{y)},\ \sigma\rangle \Rightarrow \sigma \qquad \text{if } e_n(x) \neq e_n(y) \vee e_\rho(x) \neq e_\rho(y)$$
$$\langle \texttt{assume(x}\leq\texttt{y)},\ \sigma\rangle \Rightarrow \sigma \qquad \text{if } e_n(x) \leq e_n(y)$$

Figure 4. Meaning of safe commands. $\sigma = (e, \mu, \zeta)$. The functions *Store* and *Load* are defined in Figure 5.

A command $w :=$ `shared` K attempts to retrieve a pointer to a shared memory region of size K. It might return a fresh pointer, a pointer that was returned from a similar command earlier, or a null value (num, 0).

A (safe) store to the stack $*p :=_{sz} x$ removes any segments overlapping with $(e_n(p), sz)$ from the memory, and maps this cell to the contents of x. It also updates ζ, adding the cell's bytes, if a number is written, and removes them otherwise. Note that storing a pointer into a memory cell which overlaps an existing memory cell c containing a numerical value leaves the non-overwritten addresses of c in ζ.

A (safe) load from the stack $w :=_{sz} *p$ tries to load the cell $(e_n(p), sz)$. If it does not succeed, w is set to have an arbitrary value and its tag is set to num if ζ assures us that the read addresses do not contain pointers, or fragments of, and to inv otherwise.

Loads from any other region return an arbitrary numerical value. A store to any other region has no internally visible effect. (Recall that in our semantics we assume that the pointers to the beginning and the end of the packet are stored in immutable registers and not in the context.)

The meaning of `assume` commands is straightforward. Note that pointer equality holds only if they point to the same region. Recall that pointers can only be compared to other pointers in the same region or to zero. In particular, a safe comparison between a pointer and a numerical value never holds because the regions are distinct.

$$\mathit{Store}(\mu, \zeta, c, (R, n)) = (\mu'[c \mapsto (R, n)], \zeta')$$
$$\text{where } \mu' = \mu[c_o \mapsto \bot \mid c_o \in \mathsf{Cell} \wedge \overline{c_o} \cap \overline{c} \neq \emptyset]$$
$$\zeta' = \text{if } (R = \mathsf{num}) \text{ then } (\zeta \cup \overline{c}) \text{ else } (\zeta \setminus \overline{c})$$
$$\mathit{Load}(\mu, \zeta, c) = \text{if } (c \in \mathrm{dom}(\mu)) \text{ then } \{\mu(c)\} \text{ else } (\{R'\} \times \mathbb{Z})$$
$$\text{where } R' = \text{if } (\overline{c} \subseteq \zeta) \text{ then num else inv}$$

Figure 5. Helper functions for load and store commands.

5 Static Analysis

In this section, we describe a static analysis that conservatively verifies that an eBPFPL program is *safe*. The analysis is parametric: It uses a numerical domain $\mathcal{D}_N$ to abstract numerical values and a *tag domain* $\mathcal{D}_T$ to abstract bounded sets. The former is used to conservatively track the numerical values and offsets stored in variables and memory cells and the latter to conservatively track their tags.

We define the abstraction in two steps: First we abstract the tags of pointers to shared regions by the sizes of the regions they point to. This bounds the number of possible tags in any program P. We then abstract the resulting states by applying the numerical and tag domains to obtain an effective static analyzer.

In the rest of this section, we assume to work with a fixed arbitrary program P and size map *sizeof*().

5.1 Abstracting Shared Regions

Our first step in the abstraction is replacing each shared region with its size. We denote the set of *abstract tags* of P by $\mathcal{T} = \mathcal{T}_{\mathsf{Shared}} \cup \{\mathsf{ctx}, \mathsf{stk}, \mathsf{pkt}, \mathsf{num}, \mathsf{inv}\}$, where $\mathcal{T}_{\mathsf{Shared}} = \{K \mid (w := \texttt{shared}\,K) \in P\}$. Note that the $\mathcal{T}$ is similar to $\mathcal{R}$, except that it replaces the (unbounded) set of shared region identifiers found in $\mathcal{R}$ with the (bounded) set of the *sizes* K which appear in `shared` K commands in P.

Memory states with abstract tags. The set of *machine states with abstract tags* $\widehat{\mathsf{State}}$ is similar to that of the concrete semantics except that it tags values using abstract tags $T \in \mathcal{T}$ instead of concrete tags $R \in \mathcal{R}$. For notational convenience, we also use pairs of mappings to values and tags instead of using maps to tagged values; this change does not incur information loss.

$$\begin{array}{rcll} (e_\tau, \mu_\tau) & \in & \mathsf{Tags} & = (\mathsf{Register} \to \mathcal{T}) \times (\mathsf{Cell} \hookrightarrow \mathcal{T}) \\ (e_n, \mu_n) & \in & \mathsf{Values} & = (\mathsf{Register} \to \mathbb{Z}) \times (\mathsf{Cell} \hookrightarrow \mathbb{Z}) \\ \widehat{\sigma} & \in & \widehat{\mathsf{State}} & = \mathsf{Tags} \times \mathsf{Values} \times \mathsf{Format} \end{array}$$

We define an abstraction function $\beta \in \mathsf{State} \to \widehat{\mathsf{State}}$ which replaces shared region tags with abstract tags:

$$\beta(e, \mu, \zeta) = ((e_\tau, \mu_\tau), (e_n, \mu_n), \zeta), \text{ where}$$
$$e_\tau = \begin{cases} \mathit{sizeof}(e_\rho(x)) & e_\rho(x) \in \mathsf{Shared} \\ e_\rho(x) & \text{otherwise} \end{cases}$$
$$\mu_\tau = \begin{cases} \mathit{sizeof}(\mu_\rho(x)) & \mu_\rho(x) \in \mathsf{Shared} \\ \mu_\rho(x) & \text{otherwise} \end{cases}$$

Transitions with abstract tags. The transition relation over machine states with abstract tags is a direct adaptation of the concrete transition relation to use abstract tags. This entails few minor changes. (We keep using the same notations as in Section 4, for brevity.)

Firstly, the Safe() predicate needs to perform bound checking using abstract tags. This poses no issues, as we can translate any (unbounded) map *sizeof* to a bounded *abstract map* $\widehat{sizeof}(T) = \text{if } T \in \{\text{ctx}, \text{stk}\} \text{ then } sizeof(T) \text{ else } T$.

Note that this change does not lead to more conservative checks regarding potential memory safety violations, since the size of every region is still being tracked precisely.

Secondly, as we can no longer tell whether two pointers to a shared region of size K point to the same region or not, we strengthen Safe() to forbid subtraction and less-than comparison between such pointers.

$$\begin{aligned}\mathsf{Safe}_{w\,:=\,x-y}(\widehat{\sigma}) &= e_\tau(x) = e_\tau(y) \wedge e_\tau(y) \notin \mathcal{T}_{\mathsf{Shared}} \\ &\quad \vee e_\tau(y) = \mathsf{num}. \\ \mathsf{Safe}_{\mathsf{assume}(x\,\leq\,y)}(\widehat{\sigma}) &= e_\tau(x) = e_\tau(y) \wedge e_\tau(y) \notin \mathcal{T}_{\mathsf{Shared}}\,.\end{aligned}$$

Finally, we need to weaken the filtering done by checking inequalities, since two pointers to shared regions might not be equal, even if they have the same offset and abstract tag. We do so by *adding* the following transition:

$$\langle \mathtt{assume(x{\neq}y)}, \widehat{\sigma}\rangle \Rightarrow \widehat{\sigma} \quad \text{if } e_\tau(x) = e_\tau(y) \wedge e_\tau(y) \in \mathcal{T}_{\mathsf{Shared}}\,.$$

Definition 5.1. A state $((e_\tau, \mu_\tau), (e_n, \mu_n), \zeta) \in \widehat{\text{State}}$ is *admissible* if

(i) $\text{dom}(\mu_\tau) = \text{dom}(\mu_n)$, and
(ii) $\forall c \in \text{dom}(\mu_\tau).\mu_\tau(c) \neq \mathsf{num} \rightarrow \zeta \cap \bar{c} = \emptyset$.

Our concrete semantics as well as the one we just defined does not produce arbitrary states; it ensures that the memory does not contain overlapping cells and that no addresses containing pointer values can be partially read.

Lemma 5.2. *If* $\langle cmd, \widehat{\sigma}\rangle \Rightarrow \widehat{\sigma}'$ *and* $\widehat{\sigma}$ *is admissible then* $\widehat{\sigma}'$ *is also admissible.*

Lemma 5.3. *If* $\sigma \in$ State *is a reachable state of* P *then* $\beta(\sigma)$ *is admissible.*

Thus, in the following, unless stated otherwise, we redefine $\widehat{\text{State}}$ to consider only admissible states.

We say that P is *safe to execute with abstract tags* if no execution of P according to the transition relation over machine states with abstract tags starting at a state $\beta(\sigma)$, where $\sigma \in$ State is an initial state, produces the error state.

Lemma 5.4. *If* P *is safe to execute with abstract tags then* P *is safe.*

5.2 Bounded Abstraction

The abstract interpretation algorithm computes an over-approximation of the reachable states of a program when it executes over machine states with abstract tags. We construct our analysis in a parametric manner on top of a numeric domain $\mathcal{D}_N$ and a tag domain $\mathcal{D}_T$, under the assumption that these domains come equipped with abstract transformers capable of handling variable-manipulating programs. (We make our assumptions more precise in the following.)

The main challenge we face is the need to handle load and stores operations. Our solution is to maintain a variable for every one of (the finite number of) possible cells in the memory, and instantiate the underlying domains to track the values as if every cell is a (syntactic) *analysis variable*:

$$\mathbb{V} = \mathsf{Register} \cup \mathsf{Cell}\,.$$

(For clarity, we write $v_{(a,sz)}$ when treating (a, sz) as a variable.) The tricky part of the encoding is the need to account for overlapping stores and unaligned loads, as unlike in standard variable manipulating programs, assigning a value to a cell (e.g., $v_{(a,4)}$) may affect values of other cells (e.g., $v_{(a,2)}$).

Assumptions. Before describing our analysis, we list our assumptions regarding the underlying parametric domains.

We expect $\mathcal{D}_N$ to be equipped with a least upper bound operator $\sqcup_N$, and, if necessary to ensure termination, a widening operator ∇_N. The tag domain $\mathcal{D}_T$ is required to be accompanied with a least upper bound operator $\sqcup_T$.

The numeric domain $\mathcal{D}_N$ and the tag domain $\mathcal{D}_T$ are used to abstract mappings from analysis variables to integers and (a bounded set of) abstract tags, respectively. Thus, we assume to have appropriate concretization functions $\gamma_N \in \mathcal{D}_N \rightarrow 2^{\mathbb{V}\rightarrow\mathbb{Z}}$ and $\gamma_T \in \mathcal{D}_T \rightarrow 2^{\mathbb{V}\rightarrow\mathcal{T}}$.

We expect $\mathcal{D}_N$ to come equipped with abstract transformers $[\![cmd]\!]^\sharp_N(\cdot)$ which can conservatively over-approximate assignments of arithmetic and boolean expressions to variables, and a $\mathsf{havoc}\,w$ operation (sometimes called forget) which abstracts away any information pertaining to a variable w. These requirements are quite standard. For example, the Interval, Zone, Octagon and Polyhedra domains satisfy our requirements.

The abstract transformers over the tag domain $[\![cmd]\!]^\sharp_T(\cdot)$ should support commands pertaining to variable assignments operations, assignments of constants sets of abstract tags to variables, and checking whether an analysis variable may have a particular abstract tag assigned to it.

Abstract domain. The analysis is based on an abstract domain $\Sigma^\sharp$ which is a cartesian product of a numerical domain $\mathcal{D}_N$, the tag domain $\mathcal{D}_T$, and two powerset domains, ordered by the superset relation. The first, defined over Address, conservatively tracks the addresses in the stack region containing numerical values. The second, defined over Cell, keeps track of the set of memory cells containing valid values.

$$\sigma^\sharp = (\theta, d, \zeta, \delta) \in \Sigma^\sharp = \mathcal{D}_T(\mathbb{V}) \times \mathcal{D}_N(\mathbb{V}) \times 2^{\mathsf{Address}} \times 2^{\mathsf{Cell}}$$

The first two components of the abstract state contain *may* information, namely, what may the numerical values and abstract tags of the registers and of the memory cells. The last

$$\llbracket *\mathsf{p} :=_{sz} \mathsf{x}\rrbracket^\sharp(\theta,d,\zeta,\delta) =$$
$$\begin{cases} (\llbracket v_{(a,sz)} :=_{sz} \mathsf{x}\rrbracket^\sharp_T(\theta'), \llbracket v_{(a,sz)} :=_{sz} \mathsf{x}\rrbracket^\sharp_N(d'), & \\ \qquad\qquad \zeta', \delta' \cup \{(a,sz)\}) & A=\{a\} \\ (\theta', d', \zeta \setminus \mathit{Footprint}, \delta') & 1<|A| \end{cases}$$
$$\theta' = \llbracket v_{(a,sz)} :=_{sz} \mathcal{T} \mid (a,sz) \in \mathit{Overlap}\rrbracket^\sharp_T(\theta)$$
$$d' = \llbracket \mathsf{havoc}\, v_{(a,sz)} \mid (a,sz) \in \mathit{Overlap}\rrbracket^\sharp_N(d)$$
$$\zeta' = \begin{cases} \zeta \cup \mathit{Footprint} & \theta(x) = \{\mathit{num}\} \\ \zeta \setminus \mathit{Footprint} & \text{otherwise} \end{cases}$$
$$\delta' = \delta \setminus \mathit{Overlap}$$

$$\llbracket \mathsf{w} :=_{sz} *\mathsf{p}\rrbracket^\sharp(\theta,d,\zeta,\delta) =$$
$$\begin{cases} (\llbracket \mathsf{w} := v_{(a,sz)}\rrbracket^\sharp_T(\theta), \llbracket \mathsf{w} := v_{(a,sz)}\rrbracket^\sharp_N(d), \zeta, \delta) & \\ \qquad |A| = 1 \wedge A = \{a\} \wedge (a,sz) \in \mathrm{dom}(\theta) & \\ (\llbracket \mathsf{w} := \{t\}\rrbracket^\sharp_T(\theta), \llbracket \mathsf{havoc}\,\mathsf{w}\rrbracket^\sharp_N(d), \zeta, \delta) & \text{otherwise} \end{cases}$$

$$\begin{aligned} A &= \{a \in \mathsf{Address} \mid \llbracket \mathtt{assume}(p=a)\rrbracket^\sharp_N(d) \neq \bot\} \\ \mathit{Footprint} &= \{a+i \mid a \in A \wedge 0 \le i < sz\} \\ \mathit{Overlap} &= \{c \in \mathsf{Cell} \mid a \in A \wedge \bar{c} \cap \overline{(a,sz)} \neq \emptyset\} \\ t &= \text{if } \mathit{Footprint} \subseteq \zeta \text{ then num else inv} \end{aligned}$$

Figure 6. Abstract semantics for load and store to the stack.

two components contain *must* information, namely, which of the memory cells are definitely present in the memory and which address in the stack region are used to represent numerical values.[1]

The join operator is defined in the standard way via lifting:

$$(\theta_1, d_1, \zeta_1, \delta_1) \sqcup (\theta_2, d_2, \zeta_2, \delta_2) = \\ ((\theta_1 \sqcup_T \theta_2), (d_1 \sqcup_N d_2), (\zeta_1 \cap \zeta_2), (\delta_1 \cap \delta_2)) .$$

The concretization function $\gamma \in \Sigma^\sharp \longrightarrow 2^{\widehat{\mathsf{State}}}$ maps abstract states $(\theta, d, \zeta, \delta)$ to the set of admissible machine states with abstract tags they represent. γ considers only states which agree with the values and abstract tags mappings represented by θ and d, respectively, on δ—the set of memory cells which *must* contain valid values—and in which a superset of the addresses in ζ contain numerical values:

$$\begin{aligned} \gamma(\theta,d,\zeta,\delta) = \{&(e_\tau,\mu_\tau),(e_n,\mu_n),\zeta') \in \widehat{\mathsf{State}} \mid \\ &\exists f \in \gamma_T(\theta).e_\tau = f|_{\mathsf{Register}} \wedge \mu_\tau|_\delta = f|_\delta \wedge \\ &\exists g \in \gamma_N(d).e_n = g|_{\mathsf{Register}} \wedge \mu_n|_\delta = g|_\delta \wedge \\ &\delta \subseteq \mathrm{dom}(\mu_\tau) \wedge \zeta \subseteq \zeta'\} . \end{aligned}$$

5.3 Abstract Transformers

The abstract transformers are straightforward for the Safe predicate and most of the instructions defined in Figure 4.

Figure 6 defines the abstract transformers pertaining to loading a value from the stack region or storing into it. These operations are reduced to standard variable assignments in $\mathcal{D}_N$ and $\mathcal{D}_T$, while updating the format set ζ so that it always hold only addresses that cannot possibly hold parts of pointers. We distinguish between two cases: accesses to a precisely known address and "fuzzy" accesses to location not known precisely. Such fuzzy writes may only *remove* memory cells from the memory and the set ζ.

Technically, the set A contains the addresses in the stack that p might point to. Thus, if $|A| = 1$ or $A = \{a\}$ the analysis can determine the precise address that p points to. *Footprint* contains the addresses that the operation might access, and *Overlap* all the memory cells which overlap these addresses. Note that a store operation removes any constraints on the numerical values stored in overlapping memory cells and on their abstract tags.

When the abstract tag of a pointer may be some other (valid) memory region, we perform join over all possibilities in the standard way. Such writes, like fuzzy writes, may only result in destruction of memory cells and in a smaller ζ.

We say that P is *verified to be safe* if it does not reach the error state according to the abstract transition relation.

Lemma 5.5. *If P is verified to be safe then P is safe to execute with abstract tags.*

Theorem 5.6. *If P is verified to be safe then P is safe.*

6 Verifying eBPF Programs

We implemented a prototype verifier called PREVAIL which is publicly available at [13]. PREVAIL translates eBPF binaries into a CFG-based language understood by Crab [30]—a parametric framework for modular construction of abstract interpreters. Crab provides a simple three-address instruction set that includes boolean/arithmetic/bitwise operations, gotos, assumes, assertions, and array operations. We encode abstract tags as constant numbers and used the same abstract domain to track values and tags together. We handle null checks by tracking absolute values of pointers in addition to offsets, and use a shadow array of tags for each byte in the memory.

6.1 Handling Machine Integers

Our formal description is expressed in terms of mathematical, unbounded integers. However, eBPF programs are aimed at performance; they deal with machine words, and arithmetic is defined modulo 64 bits. This has two related but distinct implications: integer arithmetic may overflow, and pointer arithmetic may overflow. PREVAIL handles both cases in a sound manner.

Integer overflow. To deal with integer overflow and underflow, we check after each arithmetic operation that the (mathematical) value of the result is representable in 64 bits. If not, the result is set to a non-deterministic value.

Pointer overflow. Pointers are tracked as offsets from some region using a numerical domain that relies on mathematical integers. This poses no problem for verifying accesses to

[1]The last component (δ) is used when validating memory safety.

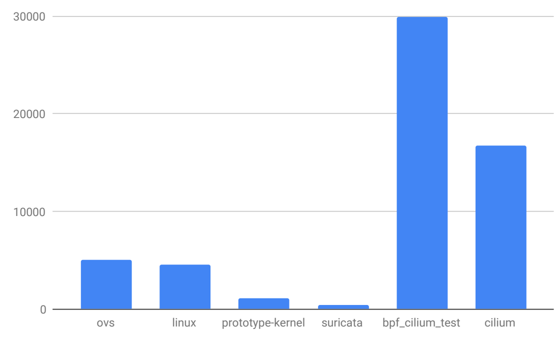

Figure 7. Total number of instructions by project.

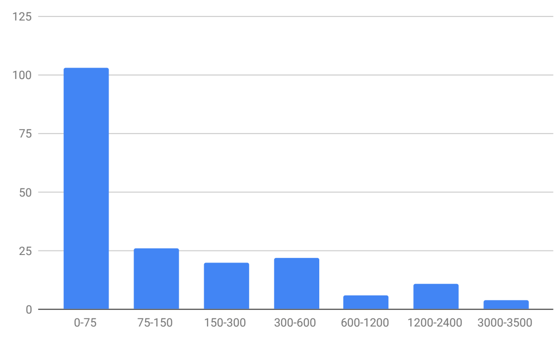

Figure 8. Number of programs (y-axis) by size ranges.

fixed-sized memory regions because if pointer arithmetics using mathematical integers produces an offset smaller than the fixed bound, so would 64 bit arithmetic.

However, the way packet accesses are handled requires comparison between the `data_end` and a potentially out-of-bounds pointer. Allowing such comparison might be unsafe, as demonstrated by the following program:

```
if (data_start + 4098 < data_end)
    *(data_start + 98) = 7;
```

This program might seem safe, but if the packet is allocated at address $2^{64} - 98$ and its size is 8, then `data_start + 4098` overflows, and the pointer comparison holds. Next, we have `data_start + 98 == NULL`, and the resulting null-dereference crashes the system.

To overcome this issue, we assume a predefined maximum size for the packet (64K, as assumed by the Linux verifier), and disallow comparisons on pointers that might reside beyond it. For example, consider the program in Table 3. To successfully verify the program the verifier needs to be able to infer that the value of `r5` is within the range $0..64K$.

6.2 Additional Supported Features

We now survey some features eBPF of programs were omitted from the formal description which our analyzer supports.

Function calls. eBPF programs utilize many predefined library functions. These functions may write to a segment of the stack given as a parameter; we model this by marking the parts of the stack they write to as containing unknown numerical values. Functions also invalidate registers `r1..r5`.

Maps. In the full eBPF language the instruction `shared` receives as an argument a special tag map, denoting the in-kernel data structure that maps keys to shared regions. Pointers to such maps are mostly loaded statically, using a special `loadFd` instruction. Tracking these maps is straightforward.

Arithmetic and division by zero. Many arithmetic operations other than + and − are supported. The analysis delegates this work to the numerical domains given as parameter. Following the existing verifier, we treat division in a special way and check for division-by-zero errors at runtime.

Bit manipulations. Bitwise operations are not tracked precisely. Instead we use efficient over-approximations, e.g., we approximate $w \mathbin{\&}= x$ (bitwise and) when $x > 0$ with $assume(w < x)$; some memory accesses rely on this property to make sure the access is within bounds. Similarly, memory writes may trim upper bits of the representation; in the implementation we track precisely only stores of full register (8 bytes), forgetting any trimmed value. eBPF also supports 32 bit arithmetic, but we did not encounter such instructions.

Unsigned comparison. Our analysis tracks signed values, yet eBPF has instructions that compare the unsigned value that is represented in a register. Instead of modeling this precisely, we allow any value to be both unsigned-less than and unsigned-greater than any other value.

Privileged programs. Some types of eBPF programs, particularly those intended at tracing, are privileged and are allowed to leak kernel information. To analyze these programs, it is enough to treat inv as if it was num in the analysis.

6.3 Unsupported Features

Our verifier does not support the following eBPF features; programs using them were removed from our benchmarks.

Map-in-map. We do not support hierarchical maps, i.e., maps which hold pointers to other maps. We encountered two programs using this feature.

Packet reallocation. We do not support changing the position or size of the packet. Support for this feature requires invalidation of all the pointers to the packet region. We encountered four programs using this feature.

Internal eBPF functions. A relatively new eBPF feature is the support of user-defined functions. We did not encounter such functions in our benchmarks, and thus implemented an intra-procedural analysis.

Miscellaneous. eBPF defines several additional program-type specific constraints, such as disallowing access to certain fields of the context or disallowing unaligned writes to certain fields. Support for these constraints has not been implemented but can be added in a straightforward way.

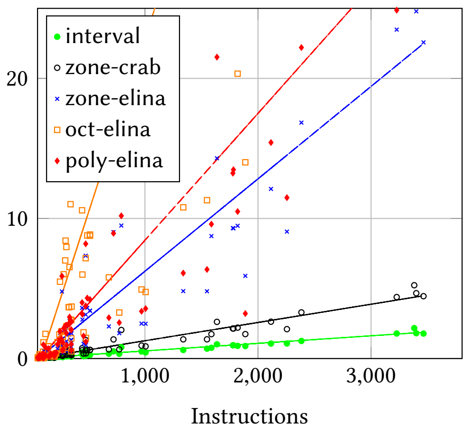

Figure 9. Analysis time (secs).

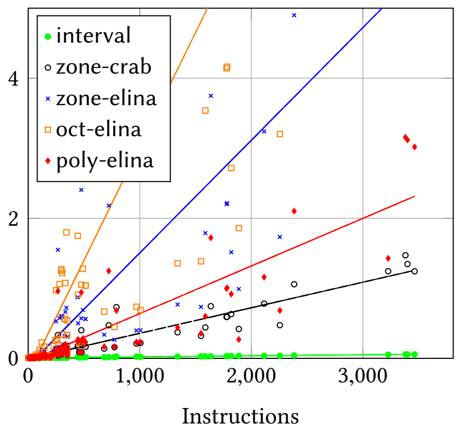

Figure 10. Memory usage (GB).

7 Empirical Results

Our evaluation aims to characterize the verification algorithm described in this paper in terms of: (1) Precision, i.e., the ability to verify real-world eBPF programs with few false positives; (2) Efficiency and scalability; (3) Handling of programs with loops.

We demonstrate on a large set of real-world eBPF programs that our verifier is at least as precise in practice as the current Linux verifier, despite giving up path information. Furthermore, it is able to correctly verify programs where the Linux verifier returns false positives. We show that in spite of its simplicity, our analysis is adequate for verifying eBPF programs, given a suitable numerical abstract domain. For the latter, we show that the Zone domain is sufficiently powerful, just like more costly domains such as Octagon and Polyhedra. We were able to verify all but one of the eBPF programs we have found in the wild within a few seconds. Furthermore, our verifier enjoys better asymptotic scalability than the Linux verifier. Finally, we apply our tool to successfully verify several programs with loops.

The experiments were performed on kernel 4.19, using a PC with a 3.40GHz Intel Core i7 CPU and 32GB of RAM.

7.1 Benchmarks

We used a set of 192 programs from six projects: linux (86 programs), a collection of eBPF programs from the Linux kernel repository; linux-prototype (23 programs), which include programs of similar purpose; ovs, programs from the Open vSwitch project [9] (18 programs); suricata [11] (5 programs), an intrusion-detection system; and cilium [3] (24 programs) and cilium-tests (36 programs), a project providing in-kernel container networking. Three of these projects (linux, ovs and suricata) guided our design and implementation, and the others served as a final evaluation. The total number of instructions in each project is given in Figure 7. Our benchmark programs are available at [12].

The only non-fixed parameter in our experiments is the numerical abstract domain used to keep track of registers and memory contents. After some preliminary tests, the numerical abstract domains used in our final evaluation are:

- interval: classical Intervals [21].
- zone-crab: Zone using sparse representation and Split Normal Form [29].
- zone-elina: Zone using online decomposition [50].
- oct-elina: Octagon using online decomposition [48].
- poly-elina: Polyhedra using online decomposition [49].

The interval domain is too imprecise to be used in practice, we include it merely as a baseline. We did not include Apron domains [32] since Elina domains supersede them.

7.2 Precision of the Analysis

Zone (zone-crab and zone-elina) and Octagon (oct-elina) prove safe all but one of the 192 programs. The non-relational interval domain fails to verify 64 programs. The domain poly-elina fails to verify 21 programs where zone-crab succeeds.[2]

7.3 Verification Cost

Figure 9 shows the execution time in seconds of the fixpoint algorithm using different numerical abstract domains as a function of the number of instructions in the program. As can be seen from the plot, zone-crab is significantly faster than the other domains, except interval. The actual runtime of zone-crab is roughly linear in the number of instructions, despite its cubic worst-case asymptotic complexity.

Figure 10 shows the memory usage of the verifier,[3] as a function of the number of instructions. Admittedly, the memory consumption of zone-crab, while better than other relational domains, is still unacceptable for an in-kernel verifier. We plan to address this issue by delegating the fixpoint computation to the untrusted user, and leaving only the final iteration to a trusted in-kernel validator.

[2]This result might seem surprising, since the Polyhedra domain is more precise than both Zone and Octagon. However, the implementation uses 64 bit integers for representing the coefficients, and falls back to top when the coefficients cannot be represented precisely using 64 bit.

[3]Extracted from the resident set size.

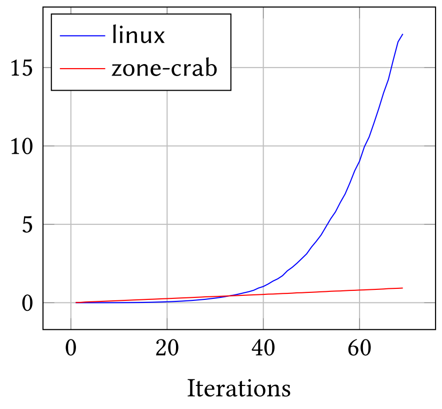

Figure 11. Execution Time (Sec) on Double `strncmp`

7.3.1 Comparison with the Linux Verifier

A fair comparison with the Linux verifier is complicated because our benchmarks are biased; these are programs that pass the verifier. Project maintainers do not publish programs that were valid but rejected (false positives), rightfully rejected (true positives) or wrongfully accepted (false negatives). The Linux verifier works by exhaustively exploring program paths, timing out after analyzing a pre-defined number of instructions (1 million in the current implementation). eBPF programs are carefully crafted to fit within this limit. It is therefore not surprising that the Linux verifier was faster than our algorithm across all benchmarks.

Next, we test our verifier on safe programs rejected by the Linux verifier due to lack of precision. We search the repositories for false positives, where the developers had to modify their code to suppress verifier errors. We found nine such commits. Our verifier was able to prove the safety in all these examples. Interestingly, some of these issues were filed as bug reports, resulting in a fix to the Linux verifier. In analyzing these fixes, we discovered that Linux relies on syntactic pattern matching and ad hoc case analysis to derive bounds on the values of program variables. For example, in one case the verifier recognized the `data + X > data_end` pattern, but not `data + X <= data_end`. It therefore does not come as a surprise that the verifier is highly fragile.

7.3.2 Verifying Programs with Loops

The Linux eBPF toolchain provides limited support for loops with static bounds by unrolling them in the compiler. This might seem sufficient given that eBPF programs must have statically bounded execution times. In practice, this proved a major pain point for developers, forcing them to do "crazy things" [20] to work around the limitation. Recall that the Linux verifier works by exhaustively enumerating program paths. A loop with N branches and i iterations yields N^i paths. We illustrate this effect using the synthetic benchmark from Example 3.4 (Section 3), where the number of paths is polynomial in `VALUE_SIZE`. As can be seen in Figure 11, the runtime of the Linux verifier grows polynomially until hitting the complexity limit at 69 iterations.

```
for (i=0; i<IPV6_MAX_HEADERS; i++){
    switch (nh) {
    case NEXTHDR_NONE: return INVALID_EXTHDR;
    ...
    case NEXTHDR_AUTH: case NEXTHDR_DEST:
        if (skb_load_bytes(...) < 0)
            return DROP_INVALID;
        nh = opthdr.nexthdr;
        len += (nh == NEXTHDR_AUTH)
            ? ipv6_authlen(&opthdr)
            : ipv6_optlen(&opthdr);
        break;
    default: ...  return len;}
}
```

(a) Skip over a chain of IPv6 extended headers.

```
for (i=0; i<ARRAY_SIZE(IPCACHE4_PREFIXES); i++){
    info = ipcache_lookup4(&map, addr,
        IPCACHE4_PREFIXES[i]);
    if (info != NULL) return info;
}
```

(b) Cache lookup (C macros expanded for readability).

Figure 12. Example loops from the cilium project.

Path explosion forces the developers to either simplify the body of the loop or pick small loop bounds to avoid the exponential path explosion. Figure 12 illustrates this using two examples from the cilium project. The first example (Figure 12a) iterates through IPv6 extended headers, determining the size of each header in order to locate the next one. It contains several branching statements, yielding multiple paths through the body of the loop. As a result, the developers had to impose an artificially low iteration bound of 4 (in reality the number of IPv6 headers is only bounded by the maximum packet size), sacrificing the ability to process packets with more headers in order to pass the verifier.

In the second example (Figure 12b), the simpler loop body allows for larger bounds (the loop bound here is equal to the size of the `IPCACHE4_PREFIXES` array); however the exact bound accepted by the verifier depends on the context where the loop is instantiated. For instance, executing multiple loops sequentially has multiplicative effect on the number of paths, thus introducing yet another exponential blowup. In fact, the developers had to establish safe bounds experimentally [1]. Recently, as the code instantiating the loop became more complicated, they were forced to reduce the size of the array at the cost of some performance degradation [2]. The eBPF community has made several attempts to introduce loop support in the verifier [28], but they did not succeed so far. In contrast, our verifier does not suffer from path explosion, as it merges paths automatically using join and widening operators. As can be seen in Figure 11, it scales linearly on the synthetic benchmark (note that in this example we deal with unrolled loops; our tool can verify this example without unrolling).

We obtain additional real-world benchmarks by searching ovs, cilium, and Linux test project repositories for commit messages indicating that a particular change was needed to overcome the verifier complexity bound. We found six such occasions, where developers refactored the code by reducing loop bounds, pushing conditional statements down in the control flow graph, etc. In all cases we were able to verify the version of the program that caused the Linux verifier to hit the complexity bound. Furthermore, verification time did not increase compared to the refactored version.

We implemented six additional tests that use loops to copy, compare, initialize the content of memory regions, compute checksums, etc. These operations frequently occur in eBPF programs, but currently only for small, fixed-size memory regions that can be handled using loop unrolling. In contrast, our examples use variable-size loop bounds. We were able to verify each of these programs in under 0.3 seconds.

8 Related Work

Securing kernel extensions. The OS community has explored numerous techniques to safely execute untrusted extensions, including the use of safe programming languages [15, 17, 27], hardware-based isolation [34, 51], and binary rewriting [46]. The main strength of eBPF is that it executes untrusted code safely with essentially zero overhead, due to the similarity with modern computer architectures. At the same time, eBPF extensions are limited in scope, as they can only be used to perform a restricted set of functions and have a very narrow interface to the rest of the kernel.

There exists a body of work on automatic verification of kernel extensions using model checking [16, 33], static analysis [4, 42], and symbolic execution [19]. While effective at finding bugs, these tools are neither sound not complete. As such, they are not applicable to untrusted extensions that may contain malicious code crafted to bypass the verifier.

Wang et al. [40, 53] present a verified compiler from BPF (*not* eBPF) bytecode to x86. Their correctness proof establishes that compiled x86 code preserves the semantics of the BPF program. It furthermore guarantees that the compiler only accepts memory-safe programs; this is straightforward, since classic BPF allows only constant-offset stack access, and packet accesses are checked at runtime.

Abstract interpretation. Abstract interpretation has been applied to prove memory safety of both high level and low level programs [23, 25, 26, 39, 43].

Astrée [18] is a static analyzer for low-level structured C code, specialized for applications such as the flight control software. Due to its huge success on real-world applications, Astrée has had a profound impact on the design and implementation of static analysis tools, including our tool.

C Global Surveyor (CGS) [52] is an array-bound checker of embedded programs such as flight control software. CGS uses pointer analysis and a numerical domain that can refine each other during the analysis. It can analyze large code bases up to 280 KLOC with 80% precision. PREVAIL targets a rather narrow class of programs, thus it does not need a pointer analysis to partition memory into disjoint regions since regions in eBPF programs can be identified statically. Furthermore, it can leverage the statically-known size of the scratch memory to reason very precisely about its contents.

Our abstraction of the stack region can be seen as a specialized version of Miné [37], which is a memory abstract domain that produces a dynamic mapping from a flat collection of abstract cells of scalar type to the set of accessed memory locations, while taking care of byte-level aliases.

Ouadjaout et al. [41] proves functional properties of device drivers in TinyOS. They precisely model the hardware state, interrupts and tasks queues. They focus on dynamic partitioning techniques [44] for achieving path-sensitivity. In contrast, our evaluation shows that path-sensitivity is not needed for precise analysis of eBPF programs.

9 Conclusions

eBPF presents a valuable opportunity for the verification community to apply state-of-the-art program analysis techniques in a domain where the need for verification is already widely accepted by developers. A verifier built on a sound theoretical foundation has the potential to dramatically simplify eBPF programming, enable new classes of programs, while providing stronger security guarantees.

Our work demonstrates that such a verifier can be built using the framework of abstract interpretation. We propose an abstraction for eBPF programs that uses Zone abstract domain adapted to track the contents of low-level memory. Our evaluation shows that the proposed abstraction is both precise and efficient for real-world eBPF programs.

Acknowledgments. This work has been supported in part by the Len Blavatnik and the Blavatnik Family foundation, Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, the Pazy Foundation, the Israel Science Foundation (ISF) grant No. 1996/18 and 1810/18, the United States-Israel Binational Science Foundation (BSF) grant No. 2016260, US NSF grants 1528153 and 1817204, and Individual Discovery Grant from the Natural Sciences and Engineering Research Council of Canada. This material is also based upon work supported by the Office of Naval Research under contract no. N68335-17-C-0558. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research.

References

[1] 2018. (2018). https://github.com/cilium/cilium/blob/master/bpf/lxc_config.h.

[2] 2018. (2018). https://github.com/cilium/cilium/commit/06efc2.

[3] 2018. Cilium: API-aware Networking and Security. https://cilium.io/. (2018).

[4] 2018. Coverity Scan: Linux. https://scan.coverity.com/projects/linux. (2018).

[5] 2018. eBPF maps. https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html. (2018).

[6] 2018. eXpress Data Path. https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/index.html. (2018).

[7] 2018. The extended Berkeley Packet Filter (eBPF) backend. http://llvm.org/docs/CodeGenerator.html#the-extended-berkeley-packet-filter-ebpf-backend. (2018).

[8] 2018. IO Visor Project. https://www.iovisor.org/technology/bcc. (2018).

[9] 2018. Production Quality, Multilayer Open Virtual Switch. https://www.openvswitch.org/. (2018).

[10] 2018. A seccomp overview. https://lwn.net/Articles/656307/. (2018).

[11] 2018. Suricata: Next Generation Intrusion Detection and Prevention Tool. https://suricata.readthedocs.io/. (2018).

[12] 2019. eBPF Benchmarks. (2019). https://github.com/vbpf/ebpf-samples.

[13] 2019. PREVAIL: a Polynomial-Runtime EBPF Verifier using an Abstract Interpretation Layer. (2019). https://github.com/vbpf/ebpf-verifier.

[14] Nadav Amit, Michael Wei, and Cheng-Chun Tu. 2017. Hypercallbacks: Decoupling Policy Decisions and Execution. In *16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. 37–41.

[15] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *16th Workshop on Hot Topics in Operating Systems (HotOS)*. 156–161.

[16] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. 2006. Thorough Static Analysis of Device Drivers. In *European Conference on Computer Systems 2006 (EuroSys '06)*. 73–85.

[17] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility Safety and Performance in the SPIN Operating System. In *Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. 267–283.

[18] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. 196–207.

[19] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 265–278.

[20] Jonathan Corbet. 2018. Bounded loops in BPF programs. https://lwn.net/Articles/773605/. (2018).

[21] Patrick Cousot and Radhia Cousot. 1976. Static Determination of Dynamic Properties of Programs. In *Proceedings of the second international symposium on Programming, Paris, France*. 106–130.

[22] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[23] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. 2009. Why does Astrée scale up? *Formal Methods in System Design* 35, 3 (2009), 229–264.

[24] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Constraints among Variables of a Program. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*. 84–97.

[25] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. 2001. Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*. 194–212.

[26] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. 2003. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. 155–167.

[27] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. 2006. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *European Conference on Computer Systems 2006 (EuroSys '06)*. 177–190.

[28] John Fastabend. 2018. [RFC PATCH 00/16] bpf, bounded loop support work in progress. https://lwn.net/ml/netdev/20180601092646.15353.28269.stgit@john-Precision-Tower-5810/. (2018).

[29] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2016. Exploiting Sparsity in Difference-Bound Matrices. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. 189–211.

[30] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 343–361.

[31] Jann Horn. 2018. eBPF memory corruption bugs. https://www.openwall.com/lists/oss-security/2017/12/21/2. (2018).

[32] Bertrand Jeannet and Antoine Miné. 2009. A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification*, A. Bouajjani and O. Maler (Eds.), Vol. 5643. 661–667.

[33] Akash Lal and Shaz Qadeer. 2014. Powering the Static Driver Verifier Using Corral. In *22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 202–212.

[34] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, Kevin Elphinstone, , and Gernot Heiser. 2005. User-Level Device Drivers: Achieved Performance. *Journal of Computer Science and Technology* 20 (2005), 654–664.

[35] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter 1993 Conference (USENIX'93)*.

[36] Antoine Miné. 2001. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Programs as Data Objects*, Olivier Danvy and Andrzej Filinski (Eds.). Vol. 2053. 155–172.

[37] Antoine Miné. 2006. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Ontario, Canada, June 14-16, 2006*. 54–63.

[38] Antoine Miné. 2006. The Octagon Abstract Domain. *Higher Order Symbol. Comput.* 19, 1 (March 2006), 31–100.

[39] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages* 4, 3-4 (2017), 120–372.

[40] MIT. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. (2014). http://css.csail.mit.edu/jitk/

[41] Abdelraouf Ouadjaout, Antoine Miné, Noureddine Lasla, and Nadjib Badache. 2016. Static analysis by abstract interpretation of functional properties of device drivers in TinyOS. *Journal of Systems and Software*

120 (2016), 114–132.

[42] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 305–318.

[43] Xavier Rival. 2003. Abstract Interpretation-Based Certification of Assembly Code. In *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*. 41–55.

[44] Xavier Rival and Laurent Mauborgne. 2007. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 26.

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

[46] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Conference on Security (USENIX Security'10)*.

[47] Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. 2000. Automatic Removal of Array Memory Leaks in Java. In *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, Arch 25 - April 2, 2000, Proceedings*. 50–66.

[48] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2015. Making numerical program analysis fast. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 303–313.

[49] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 46–59.

[50] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018. A practical construction for decomposing numerical abstract domains. *PACMPL* 2, POPL (2018), 55:1–55:28.

[51] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. 2003. Improving the Reliability of Commodity Operating Systems. In *Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. 207–222.

[52] Arnaud Venet and Guillaume P. Brat. 2004. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. 231–242.

[53] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 33–47. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wang_xi