Formal Verification of the Linux Kernel eBPF Verifier Range Analysis

Sanjit Bhat
The University of Texas at Austin
sanjit.bhat@gmail.com

Hovav Shacham
The University of Texas at Austin
hovav@cs.utexas.edu

May 2, 2022

Abstract

eBPF has emerged as one of the most important technologies in the Linux Kernel. Fundamenally, it allows userspace programs to run in kernel space, without re-compiling the kernel. While this has a number of use cases, it also creates a significant surface for privilege escalation attacks. To mitigate the security risks, Linux has a component called the eBPF verifier that checks if an eBPF program is safe to run. However, the verifier itself is over 14k lines of code, and it undergoes frequent changes. If it mispredicts an analysis, it could allow malicious programs to run in kernel space. This exact technique was used in a recent zero-day exploit that went all the way from an incorrect range analysis pass to root privilege escalation. In this work, we target the problem of a potentially faulty eBPF verifier by creating an automated formal verification framework that finds verifier bugs. Our framework takes in the raw verifier source code, translates it down to SMT using a model of the C language, combines it with correctness invariants for range analysis ops, and verifies that the source code implements a correct range analysis pass. We tested our framework by running it on dozens of security-critical commits from the past history, as well as the current commit. Using our framework, we identify where bugs where introduced and patched, we develop exploits for prior bugs, and we also shed light on a bug in the latest commit.

1 Introduction

Extended Berkeley Packet Filter, better known as eBPF, allows user processes to supply programs that execute within the Linux kernel and extend its functionality [Dev21b]. Unlike its predecessor and namesake, the BSD Packet Filter [MJ93], eBPF is used not only for packet filtering but also for security monitoring, troubleshooting and introspection, and performance profiling [Gre19]. Linux's eBPF technology is widely used in production, e.g., at Cloudflare [Maj19], Google [JM20], Facebook [SD18], and Netflix [TACL21].

The kernel's eBPF runtime provides a system call API by which user processes provide the kernel with eBPF programs to execute, specify the events that trigger program execution, and bind memory regions (called maps) to the program to facilitate input from and output to userland. The eBPF programs themselves are written using an assembly-like bytecode that supports basic constructs like registers, arithmetic operations, control flow, loops, restricted function calls, and restricted memory access. The BPF runtime either interprets these programs or (more commonly) compiles them to native machine code using a just-in-time (JIT) compiler.

The eBPF runtime must keep eBPF programs from reading and writing arbitrary kernel memory or jumping to arbitrary kernel code, even while executing those programs in the kernel address space. The simplest approach, of inlining checks before every potentially unsafe eBPF program operation, would impose an unacceptable performance overhead. Accordingly, to the greatest extent possible, the runtime checks the safety of eBPF programs at *load time* using a verifier. For example, the eBPF verifier checks that an eBPF program will always terminate, to prevent a buggy (or malicious) eBPF program from causing the kernel to hang. The eBPF JIT and verifier constitute critical attack surface. Bugs in these components can lead (indeed, have led) to exploitable privilege escalation attacks against Linux systems. (And the first demonstrated Spectre exploit relied on the eBPF JIT [Hor18].)

In this paper, we focus on a particularly complicated and bug-prone part of the eBPF verifier: the *range analysis* pass, whose job is to bound the value held in each register at each program point. If the range analysis concludes that a pointer must hold an address the program is allowed to access, then the JIT can produce code that dereferences that pointer without a runtime bounds check. A bug that leads the range analysis to conclude *incorrectly* that a pointer must be in-bounds can be abused by an attacker for read or write access to potentially arbitrary kernel memory. As an example, in the Pwn2Own 2020 competition, Manfred Paul exploited a zero-day bug in Linux's eBPF range analysis [Pau20] to find his process' `cred` struct in kernel memory and overwrite its `uid` field to 0, escalating process privilege to root.

The eBPF verifier is large, complex, and security critical. Formal verification techniques could help us gain assurance that it is free of security-relevant bugs. But the eBPF verifier also changes rapidly: See Chaignon [Cha19] for a rate-of-change analysis. A one-time, monolithic formal verification effort would become obsolete almost immediately; updating the proof artifact whenever the kernel eBPF implementation gains new features would require the kernel developers to become experts in Coq as well as C.

In this paper, we show that it is possible to gain assurance in the correctness of the Linux eBPF verifier's range analysis pass; to keep up with rapid changes to the verifier; and to use the C source of the verifier *in situ* as the basis for verification.

We build a framework that can take in an arbitrary commit of the Linux kernel, written in C, and translate the eBPF verifier range analysis pass down to logical statements. With user-specified correctness specifications, we pass the resulting verification conditions down to an SMT solver, which tells us if the C functions were correct or not. Modeling the C constructs used in this part of the kernel codebase requires handling complicated control flow and tricky integer type conversion and promotion rules that did not arise in earlier related efforts such as VeRA [BRN+20].

Our framework allows the Linux kernel developers to check the correctness of patches to the range analysis code themselves, without being reliant on verification experts to approve each change. Ultimately, we hope that our framework pays for itself by enabling developers to ship more aggressive range analyses and speed up eBPF program execution, confident that these range analyses are bug-free.

To demonstrate the scalability and general applicability of our framework, we use it to analyze dozens of versions of the range analysis code extracted from the kernel's commit history. Our framework identifies exactly where security-critical bugs were introduced, and where they were patched. These include previously acknowledged security-critical bugs like Manfred's Pwn2Own bug [Pau20], as well as other bugs not acknowledged as security-critical. One of the un-acknowledged bugs existed in the kernel for 2 months without patching; one of the acknowledged bugs existed for

14-months without patching. For each bug, our framework gives us counterexamples that we use to create exploits on real compiled kernels. Finally, we use our framework to target the latest version of the verifier. We show that while almost all functions are correct, a subtle invariant bug exists for the bitwise-And, Or, and Xor functions. We are actively working to exploit this bug.

2 Background and related work

In this section, we describe the necessary background on the eBPF language, the eBPF verifier, and formal verification to understand the rest of the paper.

2.1 eBPF language

eBPF stands for the extended Berkeley Packet Filter. As per its name, it was originally intended for packet filtering. However, today it's a much more general tool that allows users to write safe programs that get run in kernel space and have special knowledge of various kernel events.

Fundamentally, eBPF looks a lot like a basic assembly language. It models a machine with multiple *registers*, and it allows users to perform basic *arithmetic* operations like addition, subtraction, and multiplication as well as *bitwise* operations like bitwise-AND, bitwise-OR, and bitwise-XOR. These operations can occur on both 32-bit and 64-bit datatypes. eBPF supports *jumps* (both conditional and unconditional), and recently, it has added support for *loops*.

As far as memory is concerned, eBPF contains a *stack pointer*, which points to a small amount of stack space. Apart from that, eBPF also has the concept of *maps*. The eBPF runtime allows users to select from a set of pre-defined map types, which each consist of different data structures. Programs can use load and store instructions to read and write data to the map. This data can then be read by the users who submit the eBPF program. The eBPF runtime closely tracks the registers used to load and store data from maps. E.g., it marks registers used to load information as pointers to some data type in the map. This allows the runtime to check for properties like no out-of-bounds map accesses.

2.2 eBPF runtime

Users submit eBPF programs from user-space using a *system call*. In the call, they specify parameters like the map size and map type. They also define the eBPF program type, which dictates what kernel events the program can attach to. For instance, packet filter-type programs run whenever a new packet enters the system.

After the kernel receives a new eBPF program via the system call, it runs a *verifier* to guarantee program safety. Following the verification, the runtime executes the eBPF program using either a *JIT compiler*, for supported architectures, or an *interpreter*, for the remaining architectures.

2.2.1 Runtime hardening

There's relatively few dynamic protections during program execution [Dev21a]. First, for architectures that support it, the program is loaded into read-execute only memory. This prevents the program from modifying itself and voiding the results of the verifier. Second, if the `bpf_jit_harden`

flag is set, program constants are re-written to prevent them from showing up directly in executable memory. Doing so makes it harder for attackers to use eBPF programs in JIT spraying attacks. Third, unprivileged users are allowed to run `BPF_PROG_TYPE_SOCKET_FILTER` and `BPF_PROG_TYPE_CGROUP_SKB` type eBPF programs if the `unprivileged_bpf_disabled` flag is set to no. Until recently, this flag was set to no by default, but starting in November 2021, it is set to yes. Thus, for current systems, BPF only exposes a security risk if the administrator manually sets the flag to no.

Apart from these defenses, eBPF programs have no other dynamic protections. They are executed as kernel threads, and all kernel threads run within the same process in Linux. Therefore, eBPF programs technically have access to the entire kernel address space. If the verifier mispredicts an analysis and allows a malicious program, an attacker could have read/write access to arbitrary kernel memory.

2.2.2 Verifier

Due to the limited runtime defenses, the eBPF verifier plays an important role in guaranteeing safety. It checks several properties, including that the program will terminate, that it only calls designated functions, etc. The property we're most concerned with is that all memory accesses are to restricted memory regions.

To understand this property a bit more, consider the following example. Note that while this is a C example, the same logic applies to eBPF programs on registers. In fact, the main way of writing eBPF programs is by translating them from a language like C to LLVM (through something like the Clang compiler), and using LLVM's eBPF target architecture backend. In the example, imagine we have a pointer to an integer array, `arr`. We use the variable `i` to index into the array, and we need to convince the verifier that `i` lies within the bounds of `arr`:

```
int* arr;  // Initialized elsewhere
int i;     // Initialized elsewhere
int val = arr[i];
```

Earlier in the program, we have some information that helps the verifier perform its analysis. For instance, if we know the array length is less than 10, we might surround the array access in a bounds check on the variable `i`. Another way to guarantee that `i` is small is to create it from another variable with arithmetic or bitwise operations. For instance, if we create `i` by bitwise-AND'ing another variable with 0x7, we know that `i` is less than 8, which means that it's in-bounds. These two strategies are shown below, and the verifier should accept both.

```
int j;     // Initialized elsewhere
if (i < 10) {
    int val = arr[i];  // Known: i < 10
} else {
    i = j & 0x7;
    int val = arr[i];  // Known: i < 8
}
```

2.2.3 Range analysis

While the above example used either a single bounds check condition or a single bitwise-AND operation to prove an in-bounds access, this isn't always the case. There could be a number of branches and operations that all, combined, prove that an array access is in-bounds. The verifier simulates all branches and tracks information about all variables. Upon a memory access, it consults its computed information to see if the memory access is valid. In this section, we go into more depth on the range analysis information tracked by the verifier. The analyzer can be found in in the root Linux directory under `kernel/bpf/verifier.c`. In this exposition, we refer specifically to the version in commit 0f55f9ed on December 26, 2021, although many higher-level details stay the same across different commits.

Memory stores and loads use a register to specify the memory address. For every register, the verifier stores information about its possible range in a struct called `bpf_reg_state`. Each reg state has unsigned/signed 32/64-bit minimum/maximum values named like `s32_min_value`. Storing this level of information helps the verifier derive more precise bounds on variables. For instance, a signed range from -1 to 0 represents a total of 2 values, while the corresponding two's complement unsigned representation spans the entire range of 64-bit unsigned values.

In addition to the signed and unsigned ranges, registers have a `tnum` struct that encodes bit-level information about the register. Tnum's consist of two fields, a `mask` and a `value`. In a specific bit position, a `mask` has a 1 if that bit has an unknown value, and a 0 if that bit has a known value. Correspondingly, a `value` has a 1 if that bit has a known 1 value, and a 0 if that bit is either unknown or has a known 0 value. Together, these two fields in the `tnum` struct tell which individual bits are either unknown or known to be 0 or 1.

The eBPF verifier updates the range structures on conditional branches as well as on operations. For operations, the updates happen inside `adjust_scalar_min_max_vals`, which calls range analysis functions for addition, subtraction, multiplication, bitwise-AND, bitwise-OR, bitwise-Xor, bitwise-LSH (left shift), bitwise-RSH (logical right shift), and bitwise-ARSH (arithmetic right shift). The difference between logical and arithmetic right shift is that logical just shifts the bits, while arithmetic additionally copies the sign bit. Each of these functions computes updates to the reg state objects, and makes sub-calls to tnum functions inside `kernel/bpf/tnum.c` to handle `tnum` updates.

All operator range functions take in two registers and contain logic to deal with updating the output range based on various cases. For instance, in the addition function (`scalar_min_max_add`), there's a special case when the addition overflows the registers. In this case, the ranges are expanded to the maximum possible range, which is the default fail-safe operation for any range analysis operation. Doing so is always safe because it says the output range can be anything, but it also leads to a less useful output since we have less precise information about the bounds of a register. For more precise output ranges, the kernel requires tricky edge case analysis.

With five different types of ranges (signed 32-bit min/max, unsigned 32-bit min/max, signed 64-bit min/max, unsigned 64-bit min/max, and tnum), a natural question arises as to what happens if the ranges start to diverge from each other. To address this, Linux has four different functions that are often called at the end of range analysis operations. `__update_reg_bounds` improves the signed and unsigned ranges based off the corresponding `tnum`; `__reg_deduce_bounds` uses information from the signed range to update the unsigned range, and vice-versa; and `__reg_bound_offset` improves the `tnum` range based on the unsigned range.

2.3 Exploiting range analysis bugs

Due to the complexity of the ranges and machine arithmetic operations, there's often tricky corner cases in the range analysis functions. When these corner cases are incorrect, it's possible that bugs enter the codebase unnoticed. That's exactly what happened with CVE-2020-8835, when a bug in the eBPF verifier range analysis allowed a full local privilege escalation exploit [Pau20].

The zero-day involves the following `__reg_bound_offset_32` function. This function, present in an earlier verifier version but not the current, improves the `tnum` range based on the unsigned range. For context, `tnum_range` outputs a tnum representing the range of values spanning two inputs; `tnum_cast` casts a tnum to a given number of bytes by zero'ing the higher bits; `tnum_lshift` and `tnum_rshift` represent left and right shifts on tnums; `tnum_or` is the bitwise-OR of two tnums; and `tnum_intersect` is the logical intersection of two tnum ranges. Also, tnums are stored in `bpf_reg_states` using the `var_off` field.

```
static void __reg_bound_offset32(
  struct bpf_reg_state *reg)
{
  u64 mask = 0xffffFFFF;
  struct tnum range = tnum_range(
    reg->umin_value & mask,
    reg->umax_value & mask);
  struct tnum lo32 = tnum_cast(
    reg->var_off, 4);
  struct tnum hi32 = tnum_lshift(
    tnum_rshift(reg->var_off, 32), 32);

  reg->var_off = tnum_or(
    hi32,
    tnum_intersect(lo32, range));
}
```

Consider what would happen if `umin_value` $= 1$, `umax_value` $= 2^{32} + 1$, and `reg->var_off` is fully unknown. `umin_value & mask` $= 1$ and `umax_value & mask` $= 1$. `tnum_range` would generate a tnum that represents the completely known number 00...01. `lo32` and `hi32` represent completely open tnums in the lowest and highest 32 bits, respectively. When the final tnum is formed, `range` is a subset of `lo32`, so the lowest 32 bits are the completely known number 00...01. The highest 32 bits are still unknown.

This is an incorrect analysis because there's nothing in the input ranges that indicate that the lowest 32 bits of our final number are 00...01. Even though the lowest 32 bits of `umin_value` and `umax_value` both ended with 00...01, that doesn't mean that every number inbetween 1 and $2^{32} + 1$ also ends with 00...01.

Once we have one bug in the range analysis, we can chain that together in a larger attack to obtain local privilege escalation. Manfred Paul goes into more depth in his writeup [Pau20], but here's the gist of chaining this range bug into a full exploit: First, we load the value 2 from a map into a register. Since a map can hold arbitrary memory, the verifier doesn't know any information about what value is actually there. Second, we use two conditional branches to establish the `umin_value` and `umax_value` discussed above. Third, we do a 32-bit jump, which triggers the buggy code and

results in the verifier thinking that the last 32 bits of the register are 00...01, when in reality they are 00...10. Fourth, we bitwise-AND the register with 00...10, and right shift it by one. The verifier now thinks the output is 00...00, when at runtime it will be 00...01. Fifth, we can multiply this register by any constant. The verifier will think it's 0, while at runtime it actually holds our constant.

After this, we just have to do some basic pointer-scalar operations to bypass a runtime sanitization check. Then, we have a bounded read/write primitive to out-of-bounds memory, and we can read an out-of-bounds pointer in a nearby eBPF structure called the `bpf_map` to bypass kernel address space layout randomization (KASLR). Simultaneously, we can use the bounded write to modify a specific field in the `bpf_map`. A certain command of the eBPF syscall reads from the address at this field, which gives us an unbounded OOB read primitive. After some more hacking on a vtable object inside `bpf_map`, we can have eBPF call a custom function, which performs unbounded OOB writes. With unbounded OOB reads and writes, we can now walk the process credentials structure and modify our process to have root privileges.

Thus, a single flaw in an eBPF verifier range analysis function leads to local privilege escalation. Local differs from remote in that the attacker already has to have unprivileged access to the system to provide an eBPF program. This is unlike JavaScript remote privilege escalation attacks, which don't need direct attacker access because an unknowing user downloads malicious JavaScript from the web.

2.4 Protecting against range analysis bugs

The usual response to software bugs is to require more rigorous testing. However, there's good reason to believe that testing alone will not solve all the issues with the eBPF range analysis.

First, range analysis bugs are very tricky to find. Especially in the case of Linux's range objects, which are exceedingly complex. Second, these range bugs are incredible expensive. As shown in Section 2.3, one range bug can be turned into a full local privilege escalation exploit. In the remaining section, we'll consider alternates to manual testing and motivate our approach of using formal verification.

Fuzzing. Fuzzing involves testing the program with randomly chosen inputs. Simon Scannell created a fuzzer for the eBPF range analysis functions [Sca20]. In Simon's case, he creates programs with some number of random scalar operations and conditional branches. He then uses the resulting register to access memory. If the verifier accepts the program, all memory accesses should only modified mapped memory. If they modified external memory, then he knows he's found a bug in the range analysis. Using this approach, Simon found CVE-2020-27194, a typo bug in the `scalar_min_max_or` function.

Fuzzing can be a powerful approach in certain settings. For instance, the LLVM compiler infrastructure heavily relies on the libFuzzer library to catch subtle bugs [Dev21c]. Fuzzing is relatively easy to set up, you can easily bound its runtime, it hardly has a trusted compute base, and it can be relatively complete when the input search space is small. However, with eBPF range analysis functions, we don't think fuzzing can give enough assurance on the correctness of the code. The main reason is that the range analysis objects are quite complicated. This makes the state space huge, which complicates the efficacy of random testing. Another reason is that the eBPF range analysis functions don't solely consist of functions mapping to individual operations. Instead, they also have several functions that are called in several locations to coalesce ranges. Manfred Paul exploited one of these functions, `__reg_bound_offset32`, in his inital zero-day [Pau20]. The

range coalescing functions are called from several locations, and they aren't individually accessible. That makes it hard to test them in a fuzzing framework that works on external programs, since the coalescing function output will be confounded with other functions. Indeed, Manfred Paul discovered a second zero-day exploit in April 2021 [Leo21]. This exploit was in `__reg_combine_64_into_32`, another coalescing function, and it was present in the Linux kernel commit that Simon used for fuzzing.

Formal verification. Formal verification differs from fuzzing in that conceptually, it tests the program on all possible inputs instead of manually defined or randomly chosen inputs. To make this efficient, formal verification usually requires the program to be specified in some kind of logical or abstract domain. It also requires a formal definition of correctness called the *specification*. With these two, programmers prove that the abstract model of the program satisfies the specification.

There's a relatively wide design space for formally verified applications. Some projects, like CompCert [Ler09] and seL4 [KEH+09], do away with existing systems and propose complex formally verified compilers and operating systems in special-purpose languages like Haskell and Coq. These projects require developers to learn complicated verification toolchains, but they can be quite powerful in their proof abilities. Other projects, like SMACK [RE14], aim to give developers the ability to automatically verify their code directly in its implementation language. These projects support quick code updates since changes in the implementation language can be directly fed into the verification framework with no additional updates. They also remove much of the need for advanced developer skill sets since they use satisfiable modulo theory (SMT) solvers (e.g., Z3 [dMB08]) to automatically prove properties instead of manually proving them with tools like Coq. These tools are easier to use, but they have trouble proving more complex properties. Spanning inbetween these spaces are tools like Dafny [Lei10]. With Dafny, developers write code in a language that's very similar to a traditional implementation language. Along with the code, they provide invariants that help Dafny create verification conditions to dispatch to an SMT solver. This framework usually allows for proof abilities comparable to tools like Coq, but it requires more programmer effort than direct automated verification.

eBPF formal verification. Prior work in eBPF formal verification has mainly focused on the runtime JIT compiler and interpreter. In 2014, Xi Wang et al. introduced Jitk, a general framework for writing formally verified in-kernel interpeters and compilers [WLZ+13]. In Jitk, the authors use Coq to verify an extractor from eBPF to CMinor, a subset of C. They then rely on the existing CompCert framework for a formally verified translation down from CMinor to machine code [Ler09]. While this framework is promising, it requires a lot of manual proof effort in Coq to extend the eBPF language and add new features. It's also fundamentally limited by the CompCert compiler, which might not be as performant as the eBPF developers want. In 2020, two papers came out: JitSynth [GND+20] and Jitterbug [NGTW20]. JitSynth uses program synthesis to generate native architecture code that exactly matches source eBPF instructions. However, it's somewhat restricted in the complexity of instructions it can deal with, and it produces less efficient code than the native JIT compiler. Jitterbug is the closest the formal verification community has come to creating formally verified eBPF software that's actually run in the Linux kernel. It uses a verification language called Rosette [TB13] to model a fully-functional eBPF JIT compiler for several architectures. Rosette allows the authors to automatically verify that the JIT compiler is correct. From there, they can extract it to C and run it in the Linux kernel. Since this approach uses automated formal

verification, it favors fast development and ease of modification. However, it still requires kernel developers to update the JIT compiler model in Rosette, which isn't very practical. Indeed, several months after the Jitterbug authors worked with the Linux developers to verify their JITs, Linux had another zero-day vulnerability in the x86 eBPF JIT compiler [Kry21], most likely because they weren't actively updating and testing against the Rosette model.

eBPF verifier range analysis verification. Other than work verifying the eBPF JIT compiler, we know of only two other works that aim to verify the eBPF range analysis pass. PREVAIL is a self-contained eBPF verifier made from scratch [GAG+19]. It's written in C++, and it uses abstract interpretation to build a safe eBPF verifier by construction. This engine has worse runtime performance than the Linux eBPF verifier, but it offers better precision analysis that can accept more programs. While PREVAIL is a powerful verifier and was recently used in the Windows eBPF implementation [TG21], we consider it largely orthogonal to our work. The Linux developers have a mature existing verifier, and they actively add features to it. It's unlikely they would switch out the entire analyzer with PREVAIL since it requires having a large developer base that's well-versed in formal methods. In concurrent and independent work, Vishwanathan et al. apply automated verification to the tnum range structures used in Linux's verifier [VSNN21]. This work is closest in goal to our own but more limited in some important respects. Primarily, Vishwanathan et al. only analyze the tnum operations. These operations make up a small part of the range analysis, and they rely on basic C semantics to correctly model. In contrast, we try to verify a much larger subset of the range analysis operations, which involve C semantics like conditional operations and datatype conversions. Secondarily, Vishwanathan et al. manually translated the C code down to SMT. In contrast, we build an automated translation framework, which can scale to more commits.

Other verified range analysis. Although not in the eBPF space, a closely related work of ours is VeRA [BRN+20]. VeRA is a framework written in Haskell that takes C++ range analysis code from the FireFox JIT compiler codebase, converts it into logical statements, and verifies it using an SMT solver. We have the same high-level goal as VeRA, to automatically verify the range analysis code, except that we operate on the eBPF verifier instead of the FireFox JIT compiler. The eBPF codebase differs from the FireFox codebase in that it involves code constructs that VeRA doesn't currently support, such as dealing with fine-grained C type conversions. eBPF also uses different range objects and has different underlying operator semantics. However, we take inspiration from VeRA in the correction definitions that follow.

3 Correctness conditions

In this section, we build up to a description of what properties we are trying to prove.

3.1 Decomposing range analysis

The correctness of a final bounds check depends on the analyses made for all prior arithmetic operations and conditional jumps. In this work, we choose not to include conditional jump logic in our analysis, so we consider range analysis just made up from arithmetic operations. Here, overall correctness relies on each function being correct, since each analysis builds on-top of the previous.

It also relies on some glue code stitching the functions together, but we exclude that in our analysis for now. Below, we will provide the correctness conditions for a particular arithmetic operation.

3.2 Correctness of an operator

Consider an arbitrary range analysis function for a binary operator. This operator takes in two scalars, s_1, s_2, inside reg state ranges R_1, R_2, respectively. It produces an output range R_3, where s_3, the result of the operator on the scalars s_1, s_2, should be inside R_3.

The main question that arises is what does it mean for a scalar value s to live inside a range R. To answer this, we must consider that ranges contain signed/unsigned 32/64-bit minimum/maximum components, as well as a tnum component. The minimum/maximum components lead to one part of containment,

```
def contains_min_max(min, max, s):
    return min <= s and s <= max
```

Figure 1: The contains definition for min/max ranges.

For the tnum range, containment is a bit more complicated. s is contained in $(mask, value)$ if

```
def contains_tnum(mask, value, s):
    return ((~mask) & (value ^ x)) == 0
```

Figure 2: The contains definition for tnum ranges.

Intuitively, this says that for the bits in x that match the known values, the xor sub-expression bits will be 0, which matches the output. For the bits in x that don't match the known values, the xor sub-expression bits will be 1. In order for the final expression to be 0, the corresponding mask bit would need to be 1, indicating that this bit value is unknown.

With our containment definition for min/max ranges and tnum ranges, containment on a register is just

```
def contains_range(reg_state, s):
    return \
        contains_min_max(reg_state.s32_min, reg_state.s32_max, s) and \
        contains_min_max(reg_state.u32_min, reg_state.u32_max, s) and \
        contains_min_max(reg_state.s64_min, reg_state.s64_max, s) and \
        contains_min_max(reg_state.u64_min, reg_state.u64_max, s) and \
        contains_tnum(reg_state.var_off.mask, reg_state.var_off.value, s)
```

Figure 3: The contains definition for a full range object.

Thus, our main correctness property is: if R_1 contains s_1 and R_2 contains s_2, then R_3, the range returned by the range analysis function, must contain s_3, the result of the operation.

3.3 Well-formed condition

One problem with the above framework is that it may raise problems with the code when there aren't any. For instance, it could set `range1.umin_value` $= 2^{32}$ and `range1.umax_value` $= 0$ and compute an output range where `out.umin_value` $>$ `out.umax_value`. It's impossible for the scalar result to be contained in the output since if it was greater than the minimum, it wouldn't be less than the maximum, and vice-versa.

To prevent this, we add a well-formed condition to the predicate of our analysis, similar to VeRA et al. [BRN+20]. For the unsigned and signed ranges, the well-formed condition comes naturally:

```
def well_formed_min_max(min, max):
    return min <= max
```

Figure 4: The well-formed definition for min/max ranges.

For the tnum range, the well-formed definition is slightly more complicated. It says that for any bit position, mask shouldn't say that it's unknown at the same time that the value says it's a known 1. We represent this through a bitwise-AND. For every bit, either the `value` is 0, in which case the result is well-formed. Or, if the `value` is 1, the `mask` has to be 0 (an unknown bit) to make the result well-formed.

```
def well_formed_tnum(mask, value):
    return (value & mask) == 0
```

Figure 5: The well-formed definition for the tnum range.

The overall well-formedness condition for an entire range would thus be

```
def wf_range(reg_state):
    return \
        wf_min_max(reg_state.s32_min, reg_state.s32_max) and \
        wf_min_max(reg_state.u32_min, reg_state.u32_max) and \
        wf_min_max(reg_state.s64_min, reg_state.s64_max) and \
        wf_min_max(reg_state.u64_min, reg_state.u64_max) and \
        wf_tnum(reg_state.var_off.mask, reg_state.var_off.value)
```

Figure 6: The well-formed definition for a full range object.

In addition to well-formedness, the Linux kernel also defines a small number of basic additional predicates for certain operations in `adjust_scalar_min_max_vals`. E.g., for every operation outside addition, subtraction, and bitwise-AND, it requires that the second register is a known constant. Another invariant is that if the tnum represents a known value, it requires the min/max ranges to equal each other. We add the additional invariants into our verification process. We use the function `extra_preds` to sub-in for all the extra predicates for a certain function.

3.4 Modeling the range operation and the scalar operation

As always, we need some modeling of the range operation and the scalar operation. We use `out_range` and z in the below analysis to represent the resulting range and scalar, respectively. We will go over how we get these expressions more in a later section.

3.5 The final SMT query

Finally, the logical condition we would like to prove looks like this:

```
def correctness(range1, range2, out_range, x, y, z):
    lhs = \
        contains_range(range1, x) and \
        contains_range(range2, y) and \
        wf_range(range1) and \
        wf_range(range2) and \
        extra_preds(range1, range2)

    rhs = \
        contains_range(out_range, z) and \
        wf_range(out_range)

    return lhs -> rhs
}
```

Figure 7: The final correctness condition for the scalar addition range analysis function. `lhs -> rhs` stands for logical implication.

While we could feed this into the SMT solver by using things like logical quantifiers, we prefer to re-write this as a straightforward satisfiability query. Namely, we test the satisfiability of the negation of the implication, which is `lhs && !rhs`. When we feed this into the SMT solver, we either get that it's unsatisfiable, or we get that it's satisfiable. If it's unsatisfiable, then the function is correct. If it's satisfiable, then the function is incorrect. The SMT solver provies us a counterexample telling exactly which inputs the function either generated a non-well-formed output range or has the result of the operation outside the output range.

The final verification condition looks like above in all cases with binary arithmetic operators. We also verify some unary functions involved in range coalescing. These function have contains and well-formed invariants on just one range. Also, the output scalar value is just the identity since these functions model sub-register information exchanges, not actual arithmetic operations.

4 Modeling C faithfully

In all formal verification work, one big challenge is accurately modeling the implementation language in terms of logical statements that can be fed into an SMT solver. In our case, all the eBPF range functions are written in C. Additionally, the semantics of eBPF operations on registers closely follow that of the C semantics. We made a design choice to use Python as our intermediate representation

because it has a nice API to the Z3 SMT solver [Dev21d]. In this section, we discuss some of the major hurdles we went through for accurately translating between these two domains.

4.1 Memory model and why other verifier's time out

In a lot of the pre-existing tools for converting C programs into SMT (e.g., SMACK [RE14] and Crux [Tom20]), one of the major challenges is dealing with the complicated C memory model. This includes the different places memory could be placed, e.g., the stack, heap, and global executable memory. Each of these memory locations has different semantics for initializing, modifying, and deleting memory. For instance, variables on the stack are created/destroyed with scopes, variables on the heap are created/destroyed with calls to `malloc` and `free`, and variables in global memory are built into the executable and can't be removed.

Another complicated aspect of the C memory model is pointers. In C, you can define pointers to any objects in memory. You can cast these pointers to different types, perform pointer arithmetic, and dereference pointers at arbitrary points. Traditionally, these C formal verification tools have to model all of these complicated operations, which slows down verification.

In our work, we note that all range analysis functions operate on local references to data. There aren't multiple concurrent functions going on at once, so memory can be viewed as a single range operation function applied to its input. These functions modify the `bpf_reg_states` in-place, and they never perform any sophisticated pointer arithmetic. All usage of pointers is just to dereference them and access specific fields of the underlying structure. Thus, in Python, we can just create reg states in the beginning and pass around references to them. Functions will update the reg states in-place, and even if they create temporary local variables, finally, they'll assign them to fields in the register. Through this observation, we remove the need to model the complicated C memory model. Instead, we just have a perspective with functions called in sequence and composition, modifying their local pointers to reg state objects. This drastically improves verification time, and is the primary reason we had to design our own modeling tool.

4.2 Datatypes

Translating C datatypes and operations into corresponding structures in Z3 is hard. In C, there's signed and unsigned variants of 8, 16, 32, and 64-bit datatypes. Any operation can involve any combination of sign and bitwidth, and the C standard [Org07] defines a set of conversions that occur for every combination of types. In Z3 bitvectors, you can specify different bitwidths, but there is no concept of signedness. Instead, for operations that have different semantics on signed and unsigned types, Z3 splits them into different operations. Whereas C allows operations between arbitrary sign and bitwidth, Z3 only allows operations on the same bitwidth bitvectors. In this complicated input/output space, we explain how our framework translates a C operation into semantically equivalent Z3 operations.

Our own types. To model the complexities of C datatypes, we decided to create our own in Python. Although we can theoretically support the entires set of C types, the Linux range analysis pass mainly uses `s32`, `u32`, `s64`, and `u64` types. Our framework deals with the following: explicit type conversion, implicit type conversion when operating between different types, and choosing the correct signed/unsigned operation. In our analysis, we closely rely on the C99 specification to formalize exactly what C does [Org07].

Explicit type conversion. Explicit type conversions occur when a programmer explicitly casts a variable of one type to a different type. For some of these conversions, there's no meaning on the level of the bits. For instance, suppose we had a `s64` number `a` that's initialized to -1. In two's complement representation (the representation for Z3 and most modern architectures), `a` would be stored as 11...11. If we cast `a` to a `u64`, the underlying bit representation stays the same. The only thing that changes is the type of operations used on this variable, and we discuss those changes in a later section.

If we cast from a 64-bit to a 32-bit number or a 32-bit to another 32-bit number, we perform a guarding operation to make sure that the variable stays within its alloted range. The guarding operation is defined as follows. Let `min` be the minimum for a particular datatype and `max` be the maximum. Whenever converting a variable `a` to a certain datatype, we do `(a & max) - (a & min)`. For a conversion to `u32`, this has the effect of zero'ing out all the highest 32 bits. For a conversion to `s32`, this has the effect of sign extending the 31st bit all the way to the 63rd bit. These effects are useful in the arithmetic right shift code. Combined, they ensure that the output of a 32-bit arithmetic right shift has the highest 32 bits zero'd out.

Implicit type conversion. While C allows operations between arbitrary types, under the hood, it implicitly converts operands to a common type. The rules for conversion are as follows. If the two operands have the same bitwidth and sign, they stay the same type. If two operands have a different bitwidth, the smaller bitwidth operand is converted to the same sign and bitwidth as the larger operand. Otherwise, that means we're left with two operands of the same bitwidth and different signs. In that case, we convert both operands to unsigned. We implement all of these type conversion rules in our custom datatype objects.

Operations. Every operation first performs the implicit type conversion rules listed above. After that, most operations are signed/unsigned agnostic. The only exceptions are relational operations and right shift. For relational operations excluding `==` and `!=`, if the common type is signed, we choose the signed variant, and vice-versa. For right shift, if we operate on a signed value, we do an arithmetic right shift, which carries the sign bit. On an unsigned value, we do a logical right shift, which inserts zeroes. For the most part, we don't try to model operations on 32-bit values since it's not used in the actual range analysis code. The only exception is arithmetic right shift. To address this, we need the sign bit in the 63rd bit position. We can get this by first left shifting 32 bits, then performing the arithmetic right shift, then logically right shifting 32 bits.

4.3 Other constructs

Apart from the above, we apply standard techniques described in other papers (e.g., Saturn [XA07] for turning control flow to straight-line code) as well as just following the C spec [Org07], to handle things like:

1. Loop unrolling.
2. Converting if and else statements, possibly nested, into straight-line code.
3. Handling early returns intermixed with if and else statements.
4. Constants in C programs, and what type they are inferred as.

5. Static typing. Converting variables back into their pre-defined type.

6. Function returns.

5 Automation

Our above framework (see Section 4) provides a way of modeling a subset of C in Python, which is then fed into the SMT solver using the Python Z3 API. Unfortunately, there's a lot of manual effort needed for things like creating if-statement conditions, maintaining early return conditions, inserting the correct guards for assignments and function calls, unrolling loops, assigning types for constants, and assigning explicit conversions for static typing. To remove all of this effort, we build an automated system that goes all the way from raw C range analysis code to Python code that can be used as input to the SMT solver. In the following section, we explain how the automation framework works.

Extracting the C code. The Linux verifier is well over 14k lines of code. In order to parse the code in later stages, we need to filter out only the parts we need. If we didn't filter out the codebase, we would have to provide definitions for all the kernel structures and kernel functions, which gets unweildy very quickly. For instance, running the C-preprocessor on the verifier alone yields a file with well over 100k lines of code.

To filter out the codebase, we define exactly what functions we need from the range analysis. This includes functions in both the verifier (`kernel/bpf/verifier.c`) and the tnum file (`kernel/bpf/tnum.c`). We have a script that takes these functions, searches for them in the respective files, and outputs just those functions into a separate file.

Preprocessing the C code. For the later parsing code to work, we need to define all types and global variables used in the filtered functions. The types don't actually have to be correct. E.g., we could type define `u64` as an `int`. All that matters is that the later parser knows what keywords are types and what aren't. To do this, we just prepend a list of types onto the filtered function.

In the C code, the only functions defined as macros are basic things like `min`, `max`, `min_t` (the `min` function with an additional parameter for a common type), `max_t` (the same as `min_t` but for maximum), and `TNUM` (a function that wraps `tnum` initialization). For most of these functions, we can simply avoid macros and manually create these functions so that the converted Python code can call it. The only exception is `min_t` and `max_t`. Our parser doesn't allow us to feed a type into a function, so we instead have to add a preprocessing phase to convert the type parameter in this function into a string.

Generating a C AST. To do our analysis, we need the program in a form that we can easily traverse and reason about. We get this using Eli Bendersky's pycparser project, which parses a C99-standard C program and generates a full abstract syntax tree (AST) for it. The AST encodes all parts of the C program.

Transforming the C AST. Now that we have our AST, we can use it to perform all the modeling steps listed in Section 4. As an example of how we implement one modeling step, let's consider the if-statement guarding. This involves generating positive and negative conditions for every single

branch and dealing with nested branches. To generate the conditions, we build a recursive descent function that keeps track of its parent condition. Whenever it sees a new condition, it generates the condition for it and additionally logically-ANDs that with the parent condition. It does all of this in a depth-first search way starting from the top-level if-statement and exploring if's before else's. At the same time it generates conditions, it also adds assignment and function call guards to a final list of new statements. While this is just one example transformation, we created several others to perform the core automatic translation between C and Python.

Unparsing the AST into Python. With the transformed AST, we can now convert it back into actual Python code. We do this by taking a sample C unparser from the pycparser and modifying it to generate Python syntax. The unparser works quite simply. It recursively traverses the C AST and generates corresponding Python code whenever it hits a token.

Adding manually defined code. We now have our core automatically translated range functions. To this, we add the other code that couldn't be translated automatically. This is only a small section of code and includes things like the `min` and `max` macros as well as the `adjust_scalar_min_max_vals` function and `fls64` function. All of these functions include annoying syntax that would have unnecessarily complicated our automation pipeline. For instance, the `min`, `max`, and `fls64` functions are all macros that would have needed expansion. The `adjust_scalar_min_max_vals` function models parts of the registers that we haven't included. It would also lead to unnecessarily adding conditional logic to our code when that code be embedded in the preconditions for the range analysis functions. For instance, the function has an if-statement for the register bitwise-OR function that expands the range to a wide-open range if the second register isn't a known value. Instead of adding this conditional logic, which hurts verification time, we just embed in the bitwise-OR precondition that the second operand register should be a known value.

Executing with symbolic values. Now, we have all the range code in Python Z3. Next, we can define symbolic range structures and pass them into the functions. Because of all our previous work with things like the datatype framework, the range structures run through the range analysis code and generate output range structures. These output structures are themselves symbolic.

Generating verification conditions. We manually define pre- and postconditions for every range analysis function. These conditions are structured as per Section 3. The preconditions are all defined on the symbolic input register. The postconditions are all defined on the symbolic output register computed above.

Interfacing with the SMT solver. After generating verification conditions, we send these to the SMT solver. There are three possible responses. First, the solver could fail to terminate, in which case we have to quit the program. Second, the solver could return unsatisfiable, which indicates that the range function is correct. Third, the solver could return satisfiable, which indicates there was a bug in the function. If this happens, the solver returns back a *model*, which has concrete assignments for all the symbolic variables. Under these assignments, the verification condition is supposed to be satisfiable.

Debugging. To aid developer debugging, we've written a Python playback framework. After getting the model back from the SMT solver, we pack the assignments back into input verification structures. Instead of being symbolic values, these are now concrete Python integers. We then run the concrete values through the range function, which allows us to step through it line-by-line and understand exactly how the concrete values change over time. We also made a nice interface for visualizing the concrete values either as signed numbers, unsigned numbers, or bits.

Lines of Code. All in all, our framework uses the following lines of code:

Component	LoC
Automatically generated verification functions	1673
Manual verification functions (e.g., `adjust_scalar`)	200
Framework verification code	1249
C preprocessing, transpyling, and unparsing	1178

6 Results

To evaluate our framework, we run it on commits from 07cd2631 (March 24, 2020), to 0f55f9ed (December 16, 2021). We chose that starting commit because before that, the verifier's range analysis functions were all stored in-line in the same function along with other C code outside the subset of C that we support. This would make automatically generating code much harder. We chose our ending commit because it's the last commit that made meaningful changes to range analysis functions. In-between these two commits, we found 32 relevant commits to study by parsing the Git patch history of the verifier and grep'ing for changes to range analysis functions.

6.1 When were bugs introduced and all bugs patched?

For our first big study, we wanted to know when bugs were introduced, and when they were patched. To perform this study, we ran our verification framework on the 32 commits above on every function. In these results, note that from the time a function had bugs to when it got fully patched, there may have been several bugs that got patched. Our patching commit is only the patch that results in no bugs leftover. We got the following results:

32-bit Add and Sub. For 32-bit Add and Sub, i.e., the versions that take in 32-bit operands, they were broken in 3f50f132 (March 30, 2020) and patched in 3a71dc36 (May 29, 2020). This patch was not assigned a CVE, to the best of our knowledge. This is a 2-month time period where the functions were vulnerable.

64-bit Add and Sub. For 64-bit Add and Sub, i.e., the versions that take in 64-bit operands, they were broken in 3f50f132 (March 30, 2020) and patched in bc895e8b (Jan 20, 2021). This patch was assigned CVE-2021-20268. This is a 10-month time period where the functions were vulnerable.

32/64-bit And, Or, and Xor. These functions were broken in 3f50f132 (March 30, 2020) and patched in 049c4e13 (May 10, 2021). This patch was assigned CVE-2021-3490. This is a 14-month time period where the functions were vulnerable.

32-bit Lsh, Rsh, and Arsh. These functions were always correct.

64-bit Lsh, Rsh, and Arsh. These functions were broken in 3f50f132 (March 30, 2020) and patched in 3a71dc36 (May 29, 2020). This patch was not assigned a CVE, to the best of our knowledge. This is a 2-month time period where the functions were vulnerable.

As can be seen, common to all sources of bugs was commit 3f50f132, a major re-write of the verifier that introduced 32-bit bounds into the range object so that ranges could be better tracked for 32-bit arithmetic operations. While this update brought better range precision, it came with a lot of bugs. This is because the update was written in haste shortly after Paul Manfred's initial eBPF verifier 0-day [Pau20].

6.2 Exploiting buggy code

For a couple of the buggy function types above, we now walk through how we exploited the functions.

General exploit framework. For every exploit, we test to make sure it works on the version prior to a patch and doesn't work on the patch version. For a specific test, we compile that full version of the kernel and run it using qemu and GDB. After bootup, we set a breakpoint on the verifier range analysis function and run an eBPF program. By manually stepping through the range analysis code, we can assert that the ranges actually have the incorrect values that we want them to have. This provides more evidence that our C modeling matches to the actual C spec.

Commit 049c4e13. In the range analysis functions for 32-bit And, Or, and Xor, there's a check to see if both the first and second operand tnums are known. In the buggy code, the function exits directly after the check. This patch adds a call to `mark_reg_known` after this check so that the min/max ranges can also be set to the known value, if they weren't before.

The root cause of the bug is that the developers assumed that the 32-bit And, Or, and Xor ops would have the tnum known to min/max known logic taken care of by the 64-bit versions, which run right before it. However, this is not necessarily true because the 64-bit functions check for constant 64-bit tnums, while the 32-bit functions check for constant 32-bit tnums.

The exploit, with register values given by our automated framework, consists of setting the second input register to having a mask = 0xFFFFFFFF00000000 and value = 0x1 and the first input register to having a mask = 0x0 and value = 0x100000002. When calling the 64-bit And operator, it will not detect a constant tnum, so it won't mark the reg as known and will instead proceed with setting the 64-bit min/max. Meanwhile, the 32-bit And will detect a constant tnum, but it won't mark the min/max ranges as known. This results in the unsigned 32-bit max getting set to 0 and the unsigned 32-bit min getting set to 1, which violates the well-formed invariant.

Commit bc895e8b. For 64-bit Add and Subtract, these functions use the functions `signed_add32_overflows` and `signed_sub_overflows` to detect if adding or subtract will cause the result to overflow. Because of a typo, these functions took in the wrong types of input arguments (signed 64-bit instead of signed 32-bit for the 32-bit overflow function, and signed 32-bit instead of signed 64-bit for the 64-bit overflow function). This meant that they detected overflows for the wrong types.

```
static void scalar32_min_max_and(struct bpf_reg_state *dst_reg,
        struct bpf_reg_state *src_reg)
{
         bool src_known = tnum_subreg_is_const(src_reg->var_off);
         bool dst_known = tnum_subreg_is_const(dst_reg->var_off);
         struct tnum var32_off = tnum_subreg(dst_reg->var_off);
         s32 smin_val = src_reg->s32_min_value;
         u32 umax_val = src_reg->u32_max_value;

        /* Assuming scalar64_min_max_and will be called so its safe
         * to skip updating register for known 32-bit case.
         */
        if (src_known && dst_known)
             return;
    }
    ...
}
```

Figure 8: The buggy version of the 32-bit And function that was patched in commit 049c4e13.

Our exploit sets specific signed min/max 64-bit bounds in the two inputs registers. After running subtraction, the function follows the wrong control path, which results in the contains invariant to no longer hold.

```
static bool signed_add32_overflows(s64 a, s64 b)
{
         /* Do the add in u32, where overflow is well-defined */
         s32 res = (s32)((u32)a + (u32)b);

         if (b < 0)
                 return res > a;
    return res < a;
}
```

Figure 9: The buggy version of the 32-bit overflow function that was patched in commit bc895e8b.

6.3 Analyzing the latest commit

We will now focus our analysis on the latest commit of the kernel that changed the range analysis, which is 0f55f9ed (December 16, 2021).

Verification times. These are the times (in seconds) it took to verify the various functions in the latest commit.

Function	Time (in seconds)
Add 64-bit	386
Add 32-bit	37
Sub 64-bit	2294
Sub 32-bit	199
And 64-bit	1212
And 32-bit	680
Or 64-bit	4529
Or 32-bit	707
Xor 64-bit	51
Xor 32-bit	15
Lsh 64-bit	91
Lsh 32-bit	12
Rsh 64-bit	23
Rsh 32-bit	6
Arsh 64-bit	24
Arsh 32-bit	10

Bug in the latest commit. We already gave away a bit of our results, but as per the first section of our analysis, all of the arithmetic range analysis functions check out in the latest commmit. The exception, which we didn't describe above, is in a subtle invariant bug with And, Or, Xor.

Recall that the eBPF verifier calls range coalescing functions at the end of most arithmetic range analysis functions. These functions combine signed, unsigned, 32-bit, 64-bit, and tnum ranges. Since these functions are always called at the end of range analysis, one would assume that the ranges are coalesced upon input. Indeed, if we assume that, And, Or, and Xor pass verification, as stated above. However, suppose that invariant doesn't hold. Then, we see that the 64-bit versions of And, Or, and Xor have always been broken in every verifier commit we studied, including the latest version.

Below, we describe the bug in more detail. One of the range coalescing functions, `reg_bound_offset`'s job is to bring information from the unsigned range to the tnum range. Without it, we could end up in a situation where the tnum range is larger than it should be relative to the unsigned and signed ranges.

This becomes a problem in the And, Or, and Xor functions. These functions first use the tnum range (which is outdated) to update the unsigned range. They then have a part of code that pulls the unsigned range info into the signed range info if the signed range indicates that we don't cross the sign boundary. However, the unsigned corresponds to the old tnum, which is out of sync with the signed range and might be wider. As a result, the signed max goes negative, and we get a non well-formed output.

There are multiple potential fixes for this bug. One is to call the range coalescing functions every time at the start of the range analysis functions in addition to the end. We must stress, though, that as of right now, we have found no way for an actual eBPF program to result in such a range coalescing inconsistency. In that sense, this attack is still hypothetical. However, if we are to get the exploit to work, it would be an exploitable bug that would have existed in the kernel for multiple years, which means that almost all running systems would be susceptible to it.

7 Discussion

7.1 Trusted computing base

We split our TCB into internal and external components.

Internal TCB All of our code for automatically translating from C to Python is in our TCB. In addition, the manual translations for functions like `adjust_scalar` are in the TCB, as well as our framework code, which includes our Python model of C datatypes.

External TCB We rely on the Pycparser library for C parsing and Python unparsing [Ben21]. We also rely on the Z3 SMT solver, which is an industry standard [dMB08].

7.2 Future work

There are a few key areas for future work.

Bug in latest commit. We want to continue hacking on this to see if it is exploitable. We have already spent some time analyzing the eBPF verifier to see if there are any control paths that don't properly coalesce ranges, but this is still in progress.

Verifying the multiplication function. One function we didn't discuss is arithmetic multiplication. No matter what we tried, we couldn't get this function to run, even when we gave it well over 48 hours. In the future, we will try specializing the inputs to the multiplication function to try to get smaller circuits. If that fails, we will have to move over to Coq verification. We spent a few weeks on this with Dafny, and we realized that even though we had bit-wise correctness lemmas, Dafny is not built to reason about bit-vectors, and it times out even for easy proofs.

Reducing our TCB. Early on in the development, we considered using LLVM to handle our C specification modeling for us. However, it seemed like too hard of a task to convert LLVM to SMT, so we moved on to our own framework.

After more literature review, we realized at the end of development that Serval might be the solution to our problems [NBG+19]. It contains a translator from LLVM IR to Rosette, which can be then compiled down to SMT. This seems like a promising approach for reducing the TCB, although there are still open problems when considering how you would keep the ranges intact enough through compilation so that you could generate verification conditions.

Verifying more of the range analysis. We focused our efforts on the arithmetic range analysis operations because we saw that they were a frequent source of bugs. However, there's still range analysis code that handles conditional jumps, and there's also glue code that stitches everything together. All of this code leads up to the final statement proving a bounds check determination given any sequence of operations. In the future, we aim to get closer to this more general statement, even if that involves considering other verifier systems designs to make this analysis more tractible.

8 Conclusion

In this paper, we presented a framework that allows for push-button automated verification of the Linux kernel eBPF verifier range analysis pass. Using our framework, we automatically analyzed over 30 commits of the eBPF verifier and developed insights on when bugs were introduced and patched. Some of these bugs were already identified as security-critical, while other were only considered mild bugs when they were actually security-critical. Finally, we applied our framework to the latest commit of the kernel and verified every range analysis function. We also showed a long-standing bug in the latest commit.

Availability

An older version of our code is available at the following link: `https://www.dropbox.com/sh/ho4a4lxh0k04tt7/AACxwmtFYK2lmmTOh2Qk5TUya?dl=0`. The latest version will be released when the full paper comes out.

References

[Ben21] Eli Bendersky. pycparser GitHub. `https://github.com/eliben/pycparser`, 2021.

[BRN+20] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. Towards a verified range analysis for JavaScript JITs. In *Proceedings of the 41th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[Cha19] Paul Chaignon. Complexity of the BPF Verifier. `https://pchaigno.github.io/ebpf/2019/07/02/bpf-verifier-complexity.html`, 2019.

[Dev21a] The Cilium Developers. BPF and XDP Reference Guide. `https://docs.cilium.io/en/v1.8/bpf/#hardening`, 2021.

[Dev21b] The Cilium Developers. eBPF. `https://ebpf.io`, 2021.

[Dev21c] The LLVM Developers. libFuzzer – a library for coverage-guided fuzz testing. `https://llvm.org/docs/LibFuzzer.html`, 2021.

[Dev21d] The Z3 Developers. z3py Namespace Reference. `https://z3prover.github.io/api/html/namespacez3py.html`, 2021.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[GAG+19] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[GND+20] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. Synthesizing JIT Compilers for In-Kernel DSLs. In *Proceedings of the 32nd International Conference on Computer Aided Verification*, 2020.

[Gre19] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. Addison Wesley, 2019.

[Hor18] Jann Horn. Reading privileged memory with a side-channel. Online: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, January 2018.

[JM20] Gobind Johar and Varun Marupadi. New GKE Dataplane V2 increases security and visibility for containers. Online: https://cloud.google.com/blog/products/containers-kubernetes/bringing-ebpf-and-cilium-to-google-kubernetes-engine, August 2020.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, 2009.

[Kry21] Piotr Krysiuk. [CVE-2021-29154] Linux kernel incorrect computation of branch displacements in BPF JIT compiler can be abused to execute arbitrary code in Kernel mode. https://www.openwall.com/lists/oss-security/2021/04/08/1, 2021.

[Lei10] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer Berlin Heidelberg, 2010.

[Leo21] Lucas Leong. CVE-2021-31440: An Incorrect Bounds Calculation in the Linux Kernel eBPF Verifier. https://www.zerodayinitiative.com/blog/2021/5/26/cve-2021-31440-an-incorrect-bounds-calculation-in-the-linux-kernel-ebpf-verifier, 2021.

[Ler09] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Maj19] Marek Majkowski. Cloudflare architecture and how BPF eats the world. Online: https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/, May 2019.

[MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter Conference*, 1993.

[NBG+19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Symposium on Operating Systems Principles*, 2019.

[NGTW20] 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 *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[Org07] International Standards Organization. C99 with Technical corrigenda TC1, TC2, and TC3 included. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf, 2007.

[Pau20] Manfred Paul. CVE-2020-8835: Linux Kernel Privilege Escalation Via Improper eBPF Program Verification. https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification, 2020.

[RE14] Zvonimir Rakamaric and Michael Emmi. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification*, 2014.

[Sca20] Simon Scannell. Fuzzing for eBPF JIT bugs in the Linux kernel. http://scannell.me/posts/ebp_fuzzing/, 2020.

[SD18] 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/, 2018.

[TACL21] Alok Tiagi, Hariharan Ananthakrishnan, Ivan Porto Carrero, and Keerti Lakshminarayan. How Netflix uses eBPF flow logs at scale for network insight. Online: https://netflixtechblog.com/how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96, June 2021.

[TB13] Emina Torlak and Rastislav Bodik. Growing Solver-Aided Languages with Rosette. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013.

[TG21] Dave Thaler and Poorna Gaddehosur. Making eBPF work on Windows. https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/, 2021.

[Tom20] Aaron Tomb. Crux: Introducing our new open-source tool for software verification. https://galois.com/blog/2020/10/crux-introducing-our-new-open-source-tool-for-software-verification/, 2020.

[VSNN21] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Semantics, Verification, and Efficient Implementations for Tristate Numbers. 2021.

[WLZ+13] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2013.

[XA07] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29, 2007.