Developing Verified Static Analyzers for Kernel Extensions: A Related Work Report

Harishankar Vishwanathan
Rutgers University
USA
harishankar.vishwanathan@rutgers.edu

1 Introduction

My research develops static analysis to enable safe extension of operating system (OS) kernels. I focus on the Extended Berkeley Packet Filter (eBPF) [7, 21], that has emerged as a new standard for extending the Linux operating system. With eBPF, developers can write custom code that can dynamically change the way the kernel functions, without needing to recompile the kernel and reboot into it.

eBPF is an in-kernel virtual machine with a custom 64-bit instruction set. Developers write programs directly in eBPF bytecode, or in a high level language like C and invoke the Clang compiler to compile the C code to eBPF bytecode. Programs are then passed to the kernel to be attached to specific hooks: attachment points available at various locations in the kernel. Before attaching the program however, an in-kernel static analyzer, called the eBPF verifier, checks the safety of the supplied bytecode. Once the program passes the safety checks, it is JIT compiled to the native architecture of the machine, and attached to the specified hook. Because eBPF programs run in kernel space as native machine instructions, they are able to achieve high performance.

The focus of my work has been on the eBPF verifier. It is paramount the static analysis in the eBPF verifier is sound and precise. However, the status quo falls short. Unsound static analysis allows malicious behaviors; exploits can use well-crafted inputs (e.g., packets) to escalate user privileges [18–20]. Imprecise static analysis rejects safe programs forcing developers to wrestle with the verifier [8, 23]. Figure 1 shows a non-exhaustive list of the various safety considerations in the eBPF verifier. My research provides soundness and precision guarantees in the value tracking analysis in eBPF verifier (top left in Figure 1).

The rest of this document offers an overview of influential prior research literature that has shaped my research endeavors. Its purpose is to outline the design considerations and key contributions of each paper while also aiming to identify potential avenues for future research.

1. Section 2 summarizes works that aim to guarantee functional correctness of compiler translations using formal methods.
2. Section 3 summarizes key ideas in program synthesis.
3. Section 4 is an overview of PREVAIL [8], a replacement to Linux's in-kernel eBPF verifier.

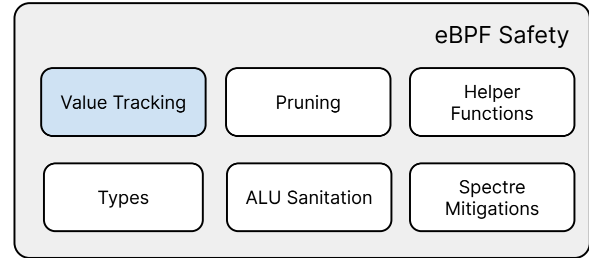

Figure 1. Safety considerations in the eBPF verifier. The focus of my work has been on the eBPF verifier's value tracking analysis.

2 Verified compilers

This section summarizes works that aim to guarantee functional correctness of compiler translations using formal methods. While these works happen to target just-in-time (JIT) compilers, the techniques have also been used in other settings. Note that in the context of eBPF, JIT compilers are "just-in-time" in the sense that a compiler within the kernel translates the eBPF bytecode down to native instructions dynamically, as it recieves it. It is not a typical JIT that interleaves interpretation and execution of compiled hot code.

2.1 Jitk

Jitk [26] targets eBPF programs that specify system call policies. In this use-case of eBPF, a developer first writes an eBPF program specifying system call policies and submits it to the kernel. The kernel then executes the program to decide whether to allow or deny a specific system call an application makes. Jitk aims to produce correct native machine code: one that correctly implements system call policies intended by the developer when executed. An additional goal Jitk has is in ensuring safety, which in this context means proving that the native machine code terminates and uses a bounded amount of stack space.

Writing eBPF code is error prone. Targetting such a specific use-case of eBPF allows Jitk to define a custom high-level specification language for this purpose (System Call Policy Language, or SCPL). Developers specify system call policies in SCPL. Being a much restrictive subset of eBPF, developers are likely to make less mistakes in specifying policies in SCPL.

The overall compilation happens in 3 phases. First, Jitk's SCPL compiler translates SCPL policies to eBPF bytecode. Next, an eBPF compiler is then invoked to translate eBPF bytecode to Cminor, an intermediate languate used in CompCert's [12] C compiler. CompCert's compilation backend finally translates Cminor to native code.

Both the SCPL-to-eBPF and the eBPF-to-Cminor compilers are written in Coq. Jitk models the semantics of SCPL, eBPF, and Cminor as state-transition systems. While Jitk needs to provide the state transition semantics for SCPL and eBPF, it borrows the one for Cminor from CompCert. Proving correctness of semantics preservation is done through a (forward) simulation of state transitions in the corresponding pairs of translations. For instance, in the eBPF-to-Cminor transltion, Jitk proves using Coq that every state transition in eBPF corresponds to some state transition(s) in Cminor.

For each compiler, Jitk's Coq source code consits of the specification, the implementation, and the a proof that the implementation matches the specification. The Coq proof checker verifies that the proof is correct. The implementation is extracted into OCaml code, and is finally converted to a native executable in the target language.

Discussion. In compilers, generally proving semantics preservation between translations involves proving backward simulation: every time the target program (T) undergoes a state transition, the source program (S) also undergoes a state transition [2]. Backward simulations are harder to prove because one needs to consider all transitions T can take, and trace them back to transitions in S. This gets tricky when T (x86 assembly) can take several transitions for one transition in S (eBPF bytecode). Using Cminor as an intermediate compilation target gives Jitk the advantage of not having to prove backward simulation in order to prove preservation of semantics when compiling down to x86. Jitk composes forward simulation proofs between SCPL to eBPF and eBPF to Cminor, and uses CompCert's proofs to build the backward simulation between x86 and SCPL. Jitk's idea of using an intermediate languages like Cminor is useful in bridging the semantic gap between eBPF and x86.

2.2 VeRA

A browser's JIT compiles and executes all the JavaScript code downloaded from the web pages during browsing sessions. These JITs are frequently exposed to malicious JavaScript code that is aimed at exploiting any vulnerabilities in the JIT. Successful exploits have allowed the code to escape the browser's sandbox, facilitating remote code execution. Typically, a compiler emits extra instructions that perform safety checks and help provide the safety guarantess that the language promises. For example, a JIT may emit instructions before an array access to validate that the array access is within bounds. While safety checks are essential for security, excessive checks can slow down browser performance. To this, address this, JITs perform static analyses to compute facts about the code that can facilitate the removal of execessive runtime checks. For instance, a static *range analysis* can be performed on the variables used as indices for array access. This analysis calculates the *range* of possible values a variable can take, at different program points. If the analysis is able to prove that the index is always within bounds (greater than 0, and less than the length of the array) before the array access, it elides emitting the extra bounds-checking machine code. VeRA [4] is a system designed to verify the correctness of these range analyses within Firefox's JavaScript JIT.

Firefox JITs range analysis routines are written in C++. VeRA provides a subset of C++ (VeRA C++) to write these range analysis routines. VeRA C++ enforces a finite semantics by disallowing loops and recursion (none of the range analysis routines in the Firefox JIT contain such constructs). VeRA C++ program is compiled to an intermediate representation (IR), built for verification: it is free of control flow, function calls, and maintains a SSA (Static Single Assignment) form for variable assignments. The IR is finally converted to SMT by a custom compiler.

Range analysis in JavaScript is somewhat more involved than settings like eBPF, because it requires reasoning about floating point numbers. An abstract value that tracks ranges of variables in JavaScript code does not simply keep track of a lower and an upper bound for that variable: it additionally tracks whether the range includes non-integrals, whether the value can be a negative zero, and so on. This also makes converting the C++ code to SMT complicated because the C++ types and operators have different semantics from the SMT counterparts. For example, the semantics of a right shift operator (>>), in C++ depends the type of the value being shifted, while SMT defines two separate operators for logical and arithmetic right shift. VeRA disambiguates the semantics when converting to IR by choosing the correct C++ operator based on the typing context. Using an IR helps bridge the semantic gap between C++ and SMT.

The authors present a verification condition that can be used to prove the correctness of range analyses, inspired by techniques in abstract interpretation [5]. A range analysis in JavaScript tracks among other things, a lower and upper bound of a program variable (a range). As the program variable undergoes operations in the JavaScript code, a range analysis must define corresponding operations that modify the upper and lower bound of the range. Consider a program point where `x` is deemed to be in the range [0, 5] and `y` is deemed to be in the range [4, 14], before the execution of the statement `z = x + y`. A *range addition* operator performs an 'addition' on the incoming range for `x` and `y`, and computes the range for the program variable `z` to be [4, 19]. Implementing a range operator might seem trivial from our example above, but an implementation quickly gets complicated when one considers other operators, overflows, and floating point values.

How to reason about the soundness of a range operator? An intuitive notion of soundness is that all possible values a program variable can take at runtime at a program point should be *contained* within the range computed for that variable at that program point (*i.e.*, it should never be the case that during some execution the variable takes a value outside the computed range). VeRA's verification condition captures this notion of containment.

Let X be the range object that tracks the range of values the variable x takes using an upper and lower bound: $X \triangleq [X.\mathtt{upper}, X.\mathtt{lower}]$. The `inRange` predicate captures the notion that all the values x can take is bounded by X (contained in X).

$$\mathtt{inRange}(X, x) \triangleq X.\mathtt{upper} \leq x \leq X.\mathtt{lower}.$$

Similarly, let Y be the range object tracking range of the variable y. The verification condition captures the idea that a sound range operator preserves containment. For a range operator $\mathtt{op}_{RA}(\cdot,\cdot)$ corresponding to the JavaScript operator $\mathtt{op}_{JS}(\cdot,\cdot)$, the verification condition is defined as follows:

$$\begin{aligned} &\forall X, Y, x, y : \\ &\quad \mathtt{inRange}(X, x) \wedge \mathtt{inRange}(Y, y) \wedge \\ &\quad Z = \mathtt{op}_{RA}(X, Y) \wedge z = \mathtt{op}_{JS}(x, y) \\ &\quad\quad \implies \mathtt{inRange}(Z, z) \end{aligned}$$

The actual `inRange` predicate is more complicated than the one mentioned above: it also defines conditions on the components of a range object other than $X.\mathtt{upper}$ and $X.\mathtt{lower}$.

The implementation of $\mathtt{op}_{RA}$ is written in VeRA C++ and extracted to SMT. The semantics of the operators $\mathtt{op}_{JS}$ is crafted by hand in SMT. Using the SMT representations in logic, the authors discharge the verification condition as a query the SMT solver. In case of a range analysis bug, the solver will return a counterexample that shows a program variable `x` not contained in its range X. Using this technique, VeRA was able to capture a bug in the `ceil` range operator in JavaScript JIT, and confirm the presence of other bugs which were fixed in the past.

Discussion. VeRA's technique of verifying range analyses has proved useful in my work. The verification condition we use to verify the soundness of the operators for the domain of tristate numbers used in the Linux kernel's eBPF verifier is similar in structure to the one specified by VeRA for the range domain [24]. In my more recent work called Agni [25], we analyzed the soundness of the Linux kernel's analysis across all the domains it uses for tracking values of program variables. Here too, we used a slightly modified version of the same verification condition.

Writing implementations of code that needs to be verified in a DSL (or a subset like VeRA C++) is a common theme to bridge the semantic gap between a higher level language and SMT, and make verification tenable. Our objective with Agni was to automaticall verify all the range analysis code in Linux across all kernel versions. Writing the kernel's range analysis for each kernel version in a DSL would be very tedious. Our approach in Agni was to rely on the LLVM IR. We wrote LLVM passes that exploit the features of the IR to convert the range analysis code to SMT, and verify it. Writing new range analysis code in a DSL and then converting it to C to be used in the kernel would likely reduce errors. However, it is difficult to integrate new know-how (DSL) and new tools (DSL compilers) into the Linux kernel's continuous integration environment. We eventually decided that using supporting a subset of LLVM IR can be sufficient to verify existing and new range analyses in the kerrnel.

2.3 Serval

While systems like Jitk use interactive theorem provers that allow verification of higher-order properties by employing richer logics, they require manual proofs and generally involve significant developer effort. A trade-off is made with so called push-button verification systems [15, 25]. These systems focus (developer effort) on (desigining) software with a finite semantics: no unbounded loops, no recursion, etc. An automated verifier symbolically evaluates the implementation to a first-order logic formula. Given a specification that the implementation must satisfy, a satisfiability query is formulated and discharged to an SMT solver to check if the implementation matches the specification.

Serval [14] aims to ease the development of the these automated verifiers, while making the approach general enough to be applicable to multiple domains (in this context, instruction sets architectures). The key idea in Serval is to create automated verifiers by writing interpreters for an instruction set that work on symbolic values. This is possible because of Rosette, an extension of the Racket programming language that eases symbolic reasoning. Developers write interpreters for concrete states in Rosette, and Rosette allows 'lifting' the intepreter to work on symbolic states. This lifted interpreter (for a particular instruction set) symbolically evaluates an implementation and reduces the semantics of the implementation to symbolic values.

An interpreter for a particular instruction set written in Rosette runs a symbolic fetch-execute-decode loop on the instructions of the program. Starting with a symbolic CPU state (*e.g.* program counter and registers), interpreting the program produces a symbolic state that encodes all possible behaviors of the program. It is important to note here that developers need to write these interpreters for each instruction set they are interested in. While it is not trivial to get this step entirely correct, one can reuse the existing CPU test suites for the specific instruction set to gain confidence. The authors of Serval provide interpreters for RISC-V, LLVM, x86-32, and eBPF instructions. Finally, the developer also provides the specification that captures the *intended* behavior of the program in Rosette. Serval uses Rosette's SMT backend to generate constraints that are then discharged to

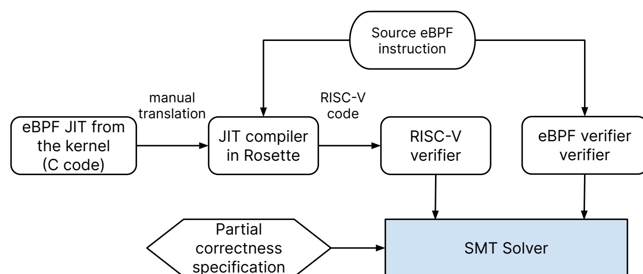

Figure 2. An instantion of Serval's pipeline for verification of eBPF JIT, considering RISC-V target.

an SMT solver to check if the implementation matches the specification.

Using Rosette allows Serval to perform some useful optimizations as it encodes the program's semantics. For instance, Rosette contains optimizations that merge two states produced by a comparison instruction: under the instruction `gtz a, b` (set `b` to `1` if `a` is greater than `0`, else set `b` to `0`), the states $(a > 0, b = 1)$ and $(a \leq 0, b = 0)$ are merged to a single state $\text{ite}(a > 0, b = 1, b = 0)$ by a Rosette optimization. Evalution can continue using this single compact encoding instead of exploring each path separately.

But now, consider the case of a jump instruction `jlt a, b, 4` (jump `4` instructions if `a` is greater than `b`). After merging the states corresponding to the program counter, the program counter pc becomes symbolic: $\text{ite}\big((a > 0), pc = pc + 4, pc = pc + 1\big)$. This a *problem* because now the interpreter has to conservatively consider all possible values of pc in order to fetch the next instruction. For large programs with nested conditionals, the number of states to explore explodes quickly. Serval addresses this by providing an optimization that works like a peephole optimization on symbolic states, borrowing from previous works in symbolic profiling [3]. First, it effectively undoes the state merging done by Rosette on pc, while keeping the other parts of the state unchanged, then evaluates the program separately on each possible concrete value of pc, and finally merges the states afterwards (on the `return` instruction).

Serval provides other such symbolic optimizations to speed up symbolic evaluation for the interpreters it provides out of the box. Writing such optimizations requires domain knowledge on the part of the interpreter developer. Developers can use Rosette's symbolic profiler to find out bottlenecks in symbolic evaluation, and come up with their own optimizations.

By combining the RISC-V (equivalently x86-32) and eBPF interpreters they built, the authors of Serval were able to write a verifier for Linux's in-kernel eBPF JIT compiler(s). Using Serval's eBPF interpreter, starting with some machine state, they symbolically execute a particular eBPF instruction to produce a resulting state. Next, using Serval's RISC-V (x86-32) interpreter, starting with an equivalent machine state, they symbolically execute the RISC-V (x86-32) instruction produced by the JIT for that eBPF instruction, to obtain a resulting state. Finally, the verifier checks if both the resulting states are equivalent. This is repeated for each eBPF instruction (they only support arithmetic and logic eBPF instructions). Using this process, they found 9 bugs in Linux's RISC-V JIT implementation and 6 bugs in the x86-32 JIT implementation.

Discussion. Serval is a demonstration of the strengths of the Rosette and Racket environment. One can write an interpreter for a concrete DSL and with the help of Rosette, lift the interpreter to work on symbolic values. With Rosette, you get symbolic evaluation, partial evaluation, and also a symbolic profiler to determine bottlenecks in verification.

The authors build interpreters for LLVM, along with ones for RISC-V, x86-32, and BPF. In my recent work on verifying C code from the Linux kernel [25], we convert the C code to logic by first converting it to LLVM IR using the clang compiler [11], in a semantics preserving transformation. We then run custom LLVM passes to convert the kernel IR to SMT. Writing the LLVM pass that compiles LLVM to SMT is an involved process, while also being to profile and optimize the solvig time when using the generated SMT for verificaiton. Currently our verification takes upwards of a day for certain kernel functions we want to verify. It could be possible to use Serval's lifted LLVM interpreter on the LLVM IR generated from the kernel's C code. This approach holds the promise of allowing our SMT formulas to be more compact, allowing verification to scale, while also affording us the opportunity to make use of Serval's (and Rosette's) optimizations and profiling if the need arises.

2.4 Jitterbug

The specification used by Serval is not complete in that it cannot be used to prove the correctness of the entire eBPF JIT. Proving the entire JIT correct for systems like the eBPF JIT is challenging because they aren't designed with verification in mind. For instance, while these JITs can be viewed as one-pass compilers, they still perform optimizations. Consider a 32-bit load from memory to a 64-bit register, where the semantics of eBPF require that the top 32-bits of the 64-bit register be cleared after the load. If an analysis can deem that the top 32-bits are already zero the JIT avoids emitting instructions that clear the top 32-bits. A correcness specification needs to reason about optimizations, along with the semantics of the source and target instructions.

Jitterbug [17] aims to address this by providing a specification that is both amenable to verification and capable of finding bugs. Jitterbug models eBPF and target architectures as abstract machines, and formulates JIT correctness as a bisimulation property between the source and target machines: a correct JIT produces a target program that preserves the behaviours of the source program. As abstract machines,

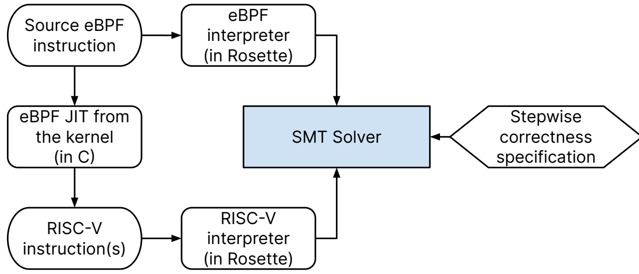

Figure 3. An instantion of Jitterbug's verification pipeline for RISC-V targets.

both the source and the target programs transition through a sequence of states producing a trace of externally visible events along the way (such as memory loads and stores, or function calls to eBPF helpers), and a return value. For a JIT to be correct, the source and target programs must produce the same trace and return value (bisimulation). Additionally, the target program must also preserve architectural safety properties (for example callee-saved registers must be preserved). Overall JIT correctness is implied if both the bisimulation and architectural safety properties hold.

Jitterbug devises a stepwise specification that implies the above notion of correctness (stepwise here means reasoning about individual translation steps). To achieve this, Jitterbug assumes (as is in the case with existing JITs in the Linux kernel), that JITs generate target programs in a per-instruction fashion. Under this assumption, for each eBPF instruction JITs emit machine instruction(s) consisting of a prologue of instruction(s), instruction(s) corresponding to the source eBPF program's instructions, and an epilogue of instruction(s). Jitterbug specifies notions of correctness for each component (prologue correctness, per-instruction correctness, and epilogue correctness). Together, these capture the idea that invariants set up by in prologue are preserved by each instruction, the trace produced by executing the machine instructions is the same as the one produced by executing the corresponding eBPF instructions, and that the epilogue results in a final state that satisfies architectural safety and produces the same return value as in the final state of the source eBPF program. Finally, the authors prove using the Lean theorem prover [6] that taken together, these correctness properties inductively imply the correctness of the entire JIT. Using a stepwise specification is also more amenable to verification as it allows working modularly to prove individual translations correct.

One of the challenges in verifying the correctness of eBPF JITs, even when following the stepwise specification mentioned above, is the dynamic nature of the code they generate. The JIT may produce varying blocks of target instructions for a given eBPF source instruction. As a particular case, consider a target architecture that has a different number of registers compared to the eBPF virtual machine. Consider also that the JIT maintains a pre-determined mapping between eBPF and target registers: if the target architecture has sufficient registers, there is a one-to-one mapping, if not, some registers can be spilled to stack. Importantly, the JIT will emit different number of target instruction blocks for a particuler eBPF instruction depending on whether the machine registers corresponding to the source eBPF registers have been previously spilled. If a register is spilled, the machine instruction block would begin with a load operation from memory. Hence, when symbolically evaluating a JIT implementation, Jitterbug produces a representation that contains *symbolic* instructions: containing symbolic values in registers and immediates. This means that unlike Serval, where the only inputs were symbolic, Jitterbug needs to reason about symbolic programs as well. which can lead to path explosion. The solution adopted by Jitterbug to address path explosion is similar to the one used by Serval to address a symbolic program counter: merge the program state at each control-flow join, and force a split on every possible concrete opcode when encountering a symbolic instruction [3].

Finally, Jitterbug specifies a DSL (in Rosette) that is a subset of C, to write JITs. Jitterbug authors manually converted the existing JITs for RV64, Arm32, Arm64, x86-32, and x86-64 in the Linux kernel to this DSL. Applying the stepwise specification from above, they attempted to prove their correctness. They found 16 new bugs in the existing JITs, and patched them. They also wrote a BPF JIT for RV32 (which did not exist in the kernel) in the DSL from scratch, and proved its correctness. They provide an (unverified) C code extractor that extracts programs in the DSL into C, which they used on their RV32 JIT implementation. Their C RV32 JIT implementation has been upstreamed into the Linux kernel.

Discussion. The main contribution of Jitterbug that stands out is coming up with the precise stepwise specification. This allows reasoning about individual translation steps of a JIT, as opposed to reasoning about the entire translation at once. This approach should work for systems that are designed modularly, as is the case with eBPF JITs.

In my recent work [25] we aim to prove the correctness of abstract interpretation of eBPF bytecode performed by the eBPF verifier. Abstract interpretation in the eBPF verifier aims to overapproximate set of possible states of the eBPF program in order to prove certain safety properties of the program. Abstract interpretation specifies a semantics on abstract values, corresponding to the concrete eBPF semantics. Establishing a correctness property for per-instruction abstract semantics is relatively straightforward, as we do in Agni [25]. However, it is curious to see if we can prove the correctness of the entire abstract interpretation similar to Jitterbug through induction (a step towards this was our refined soundness specification, which leveraged certain peculiarities of the implementation of the kernel's abstract interpretation). It is an interesting area for exploration.

3 Synthesis

This section summarizes ideas in program synthesis relevant to my work. Program synthesis is the process of automatically finding programs that satisfy some user provided specification. This process is challenging because any synthesis approach, in essence, has to search over the space of *all* possible programs, which is vast and grows exponentially with the length of the programs considered. Additionally, it is also difficult to express user intents in the form of a specification.

Program synthesis approaches can be categorized along three dimensions.

1. **The specificaton of user intent.** User intent can be specified as a logical specification, input-output examples, in natural language, or even as partial programs.
2. **The search space.** The search space should be expressive enough to include a large class of programs, while also being restrictive enough to be able to search through. Search spaces can be restricted by using only a subset of the language or using a DSL; restrictions can be placed on the operators and the control structure of the program.
3. **The search strategy.** The strategy used to search the space can be based on *enumeration* of programs in the language, *deduction* (fixing the structure of an expression, and searching for sub-expressions), *constraint solving* (translating the required behavior of the program into logical constraints that can be solved for) and even *statistical* techniques.

Consider a user intent provided in the form of a logical specification that relates inputs of the program to its expected output. For example, a specification ϕ_{spec} for a program that takes as input a list l and outputs the maximum number m could be written as: $\phi_{spec} \triangleq \forall x \in l : m \geq x$. The synthesis problem can be framed as finding a program P, that satisfies a certain specification ϕ_{spec} for all inputs I given to it and the outputs O it returns.

$$\exists P : \forall I, O : P(I) = O \implies \phi_{spec}(I, O) \quad (1)$$

The above formulation is a statement in second-order logic, as it quantifies over the space of programs. The general idea in many program synthesis approaches is to reduce the second-order search problem to a first-order search problem, that can be solved with off-the-shelf SMT solvers.

3.1 CEGIS

Counterexample-Guided Inductive Synthesis [22] (CEGIS) is based on the observation that both (a) finding a program that satisfies the specification for some inputs, and (b) verifying whether a program satisfies the specification for all inputs, are both first order problems. We can characterize these problems as follows.

1. **Finite synthesis**. Finding *one* program P_c that satisfies our specification for *some* finite number of input-output pairs $\langle I^0, O^0\rangle, \langle I^1, O^1\rangle, \ldots$.

$$\begin{aligned} &\exists P_c, I^0, O^0, I^1, O^1, \ldots : \\ &\quad P_c(I^0) = O^0 \wedge \phi_{spec}(I^0, O^0) \wedge \\ &\quad P_c(I^1) = O^1 \wedge \phi_{spec}(I^1, O^1) \wedge \ \ldots \end{aligned} \quad (2)$$

2. **Verification**. Verifying whether a program P_c satisfies a specification for *all* inputs and outputs.

$$\forall I, O : P_c(I) = O \implies \phi_{spec}(I, O) \quad (3)$$

CEGIS begins with a initial finite set S of input-output pairs that satisfy the specification: $S \triangleq \{\langle I^0, O^0\rangle, \langle I^1, O^1\rangle, \ldots\}$. It then discharges the finite synthesis query from Equation 2 to an SMT solver. The solver generates a candidate program P_c that is correct for at least all the input-output pairs in S. Next, CEGIS performs a verification of whether P_c is correct for *all* inputs, by discharging Equation 3 to the SMT solver (asking if its negation is unsatisfiable). If Equation 3 is valid, CEGIS is done, it found a program what works for all inputs and satisfies the specification. If not, the solver returns a counterexample: an input-output pair I^c, O^c for which P_c is incorrect. In this case, CEGIS adds this example to the set S, and loops back to perform finite synthesis and verification again. With each iteration of the loop, verification generates more and more counterexamples, which force the finite synthesis to come up with more and more general programs, until it finds a program that works for all inputs.

3.2 Brahma

Brahma [9] approaches the task of generating a program by breaking down a program P into components and composing them. Components are essentially functions, and are drawn from a pre-defined library. Synthesized programs are loop-free, restricted to only use components from the library, and use each component exactly once. With the idea that a candidate program can be made of components composed in an arbitrary fashion, Brahma turns the problem of synthesis into finding a location (*i.e.*, a line number) for each component in the program. Brahma generates a synthesis constraint that encodes the problem and then uses the CEGIS paradigm for solving the constraint. We first look at the structure of the library of components, followed by how the verification step of CEGIS works in Brahma, and finally the finite synthesis step.

Component library. Components are specificed using a functional description that relate its inputs to its outputs. A specification ϕ_i of a component i can look like $\phi_i(I_i, O_i) \triangleq O_0 = I_0 + 1$, which says the the output of the component is one greater than its input. Let $P \triangleq \{I_0, I_1, I_2 \ldots\}$ be the set of all inputs to all the compnents, and $R \triangleq \{I_0, I_1, I_2 \ldots\}$ be the set of all the outputs. Brahma defines ϕ_{lib} as the combination of all the component specifications in the library:

$$\phi_{lib}(P, R) \triangleq \phi_0(I_0, O_0) \cup \phi_1(I_1, O_1) \cup \dots$$

Verification. A candidate program from the finite synthesis essentially generates a mapping between the inputs and outputs of the various components. Such a mapping takes the form:

$$\phi_{conn}(I, O, P, R) \triangleq \bigwedge I_i = O_j$$

which says that the input to a component i is taken from the output of the component j, and ϕ_{conn} is a conjunction of such mappings. Note that the input to a component can be from the program input I, or from temporary results in R. The output of the program O is the output of the last component according to the mapping. The verification step in Brahma asks, given a candiate program described by location mappings ϕ_{conn} of components, does there exist some input to the program for which the specification is not satisfied?

$$\exists I, O, P, R : \\ \phi_{conn}(I, O, P, R) \land \phi_{lib}(P, R) \land \neg\phi_{spec}(I, O)$$

Finite synthesis. Now, let's look at how to generate a candidate program in the first place, using finite synthesis. Finite synthesis uses location variables to encode location mappings for components. Every component i's input I_i and output O_i parameters is associated with location variables l_{I_i} and l_{O_i} respecitvely. Each location variable determines *where* its associated parameters get its values from (this could be the program's input or from the a specific outputs of another component). The finite synthesis query searches for an assignment to these location variables.

Let L be the set of all location variables: $L \triangleq \{l_x \mid x \in P \cup R\}$. In a valid program, the location variables must satisfy some well-formedness conditions: 1. all line numbers should be within a finite range depending on the type and number of components used 2. no pair of distinct component *result* location variables must be assigned the same line number 3. line numbers assigned to location variables should be such that components only take inputs that were computed earlier. These well-formedness conditions are captured in a predicate $\psi_{wf}(L)$.

Now finite synthesis further needs to define the dataflow between components. When a component's i's output is connected to another component j's input, the result variable of component i should be constrianed to be equal to the input variable of component j. Brahma defines ψ_{conn} to capture the fact that if a pair of locations variables l_x and l_y corresponding to variables x and y of two compnents refer to the same location, then x and y must be the same value.

$$\psi_{conn}(L, I, O, P, R) \triangleq \bigwedge l_x = l_y \implies x = y$$

Finally, we can write down our finite synthesis query.

$$\exists L, O_0, O_1, \dots, P_0, P_1, \dots R_0, R_1, \dots, : \\ \psi_{wf}(L) \land \\ \bigwedge_i \phi_{lib}(P_i, R_i) \land \psi_{conn}(L, I_i, O_i, P_i, R_i) \land \phi_{spec}(I_i, O_i)$$

The synthesis query asks the solver if there exists some location mapping L for the set of inputs in the running list of counterexamples, where for each input i, the compnents are connected according to ψ_{conn} and the specification is satisfied by the ith example. When the solver finds such a location mapping, we have a candidate program to run through the verification step to see if the program is correct on all inputs. If the verification succeeds, we have found our candidate program.

4 PREVAIL

PREVAIL [8] is a replacement to Linux's in-kernel eBPF verifier built using abstract interpretation from the ground up. PREVAIL aims to improve upon the precision of the in-kernel Linux eBPF verifier by reducing the number of false positives, while also improving upon its scalability, by allowing the verification of eBPF programs with a larger number of paths (and loops).

Observations. eBPF programs can access a fixed set of memory regions: *context, stack, packet.* The stack region is a fixed-size program stack. The context region is essentially a C struct (and hence, also fixed in size). It stores invocation arguments to the eBPF program, including the start and end pointers to the packet region. The packet region, in turn, is a variable sized region that stores incoming/outgoing network packets to/from the eBPF program.

The authors make some important observations about the nature of eBPF programs that drives their choice of abstract domains used for the abstract interpretation in their verifer. First, an eBPF program performs pointer arithmetic and comparisons when accessing the variable-sized packet region. To ensure that such an access is within bounds, a verifier must *track relations between program variables (registers)* used for comparisons. This necessitates the use of a *relational* abstract domain. Next, when compilers run out of registers to use, they spill the registers by storing them to the stack region (and eventually loading them back). Proving the safety of a memory access that involves a spilled value would involve keeping track of values in memory as well. Thus, the authors note: a verifier must *track values in memory, including relations between different locations.* Finally, the authors note that to scale to being able to verify larger eBPF programs, path enumeration is not feasible. To address that, a verifier can *employ abstract domains that are equipped with joining and widening operators*: joining allows the merging of abstract states at control flow merge points, while widening allows

the convergence of abstract states in programs with loops, preventing path explosion.

A custom DSL. With the above considerations in mind, PREVAIL specifies a custom DSL (EBPFPL). eBPF programs are compiled to this DSL. The interesting aspect of the design of EBPFPL is its semantics, which is a modified form of a small step operational semantics. Given an existing state σ, the semantics of a command *cmd* in EBPFPL is defined so that the state transitions to σ' only if a safety predicate Safe(cmd, σ) is additionally satisfied, and transitions to an error state otherwise. Thus, the semantics enforces runtime statefy by aborting into an error state when it detects a safety violation. PREVAIL develops an abstract interpretation that overapproximates EBPFPL's semantics. Now all PREVAIL's abstract interpretation needs to do to prove safety of an eBPF program is to prove that the compiled EBPFPL program never aborts.

Semantics. The formalization of the rest of the semantics of eBPFPL is driven by the previously mentioned peculiarities of eBPF. PREVAIL models every program variable (which translates to a register or a memory cell in eBPF bytecode) with a tag and a value. The *tag* is chosen from one of {context, packet, stack} if the register (or memory cell) contains a pointer to one of the corresponding regions, or it is set to numerical if the register contains a purely scalar value. The *value* represents the actual value in case of a register (or memory cell) that has been tagged numerical, or an offset into the specific region in case of one tagged context or packet or stack. As an example, (stack, 10) represents a pointer 10 bytes into the stack while (numerical, 10) represents a scalar value 10.

The semantics of each EBPFPL command specifies not only the usual state transition relations, but also the command-specific Safe(cmd, σ) as predicates mentioned previously. As an example, consider the assignment command `w := E`. It is trivally safe for simple assignments of immediates or another register. However, in case `E` is of the form `x ± y`, the safety predicate captures the idea that the assignment shouldn't lead to undefined pointer arithmetics (*e.g.* adding/subtracting pointers to two distinct regions). Similarly, load (store) commands are deemed safe if they only load from (store to) bytes within the region the pointer is tagged as (stores need an additional condition for safety to disallow leakage of pointers to externally-visible memory: if the value to store is a pointer, the address at which to store can only be of a stack tag).

Abstract interpretation. PREVAIL develops an abstract interpretation that conservatively overapproximates the semantics of EBPFPL programs in order to determine their safety. Using the Crab [10] abstract interpretation framework, the authors were able to parametrize their analysis to select from one of several abstract domains. The overall abstract interpretation requires two domains for tracking the values and tags of registers: a numerical domain for tracking values and a custom domain for tracking tags. In practice, PREVAIL's algorithm encodes abstract tags as constant numbers, and use the same abstrat domain to track both values and tags. The authors evaluate PREVAIL on the non-relational interval domain, and on the relational zone, octagon, and polyhedra domains [13]. Interestingly, in abstracting *bitwise* commands, the domains used by PREVAIL are less precise than the ones used in the existing in-kernel verifier [24].

Results. The non-relational domains unsurprisingly outperform the interval domain in terms of precision (fewer false positives). However, this precision comes at a cost of increased time and space complexity of abstract operators for the non-relational domains. Consequently, as the size of the programs in terms of instructions (and number of variables to track) increases, the verification time also increases. The authors observe that the zone domain is the one that performs the best, both in terms of verification time and memory consumption.

Discussion. One of the appealing aspects of PREVAIL's design is its modularity. It should be possible to use it with new abstract domains and operators in the future. In its current form however, the main appeal for PREVAIL comes from fact that using more expressive relational domains reduces the false positve rate for a verifier. Moreover, these domains are equipped with *widen* operators, allowing PREVAIL to prove the safety of eBPF programs with loops.

However, due to the high memory consumption PREVAIL would likely not be accepted into the kernel as a drop-in replacement for the current verifier. The authors propose performing the expensive parts of the verification in PREVAIL in userspace, with a kernel component merely validating if the verification was performed correctly. This 'proof-carrying' approach has also been proposed by others [16], and is an interesting dimension to explore in the design space for securing eBPF programs with static analyses.

It is also important to note that the current in-kernel verifier performs additional analyses to mitigate certain other kinds of vulnerabilities (*e.g.* speculative execution), and it is not clear if a purely abstract interpretation based approach like PREVAIL would work for accomplishing the same [1].

5 Conclusion

My reserach involves elements of all three types of systems discussed in this report: verification, synthesis, and abstract interpretation. It has been educative to understand and learn from the specific trade-offs they've made to make their solutions work. I look forward to applying the techniques learnt in my future work.

References

[1] Nadav Amit. 2019. *Should verification be done in the kernel?* Retrieved October 25, 2023 from https://lwn.net/Articles/795037/

[2] Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2020. Towards Formally Verified Just-in-Time Compilation *(CoqPL 2020)*. https://people.irisa.fr/Aurele.Barriere/papers/coqpl20_abstract.pdf

[3] James Bornholt and Emina Torlak. 2018. Finding Code That Explodes under Symbolic Evaluation *(OOPSLA 2018)*. https://unsat.cs.washington.edu/papers/bornholt-sympro.pdf

[4] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a verified range analysis for JavaScript JITs *(PLDI 2020)*. https://doi.org/10.1145/3385412.3385968

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints *(POPL 1977)*. https://dl.acm.org/doi/pdf/10.1145/512950.512973

[6] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description) *(CADE 2015)*. https://kilthub.cmu.edu/articles/journal_contribution/The_Lean_Theorem_Prover_system_description_/6492815/1/files/11937416.pdf

[7] Matt Fleming. . *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*. Retrieved October 25, 2023 from https://www.kernel.org/doc/Documentation/networking/filter.txt

[8] 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 *(PLDI 2019)*. https://doi.org/10.1145/3314221.3314590

[9] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices* (2011). https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/pldi11-loopfree-synthesis.pdf

[10] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework *(CAV 2015)*. https://ntrs.nasa.gov/api/citations/20160001255/downloads/20160001255.pdf

[11] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation *(CGO 2004)*. http://vglab.cse.iitd.ernet.in/~sbansal/csl862-virt/readings/llvm.pdf

[12] Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning* (2009). https://doi.org/10.1007/s10817-009-9155-4

[13] Antoine Miné. 2017. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages* (2017). https://hal.sorbonne-universite.fr/hal-01657536/document

[14] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval *(SOSP 2019)*. https://unsat.cs.washington.edu/papers/nelson-serval.pdf

[15] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel *(SOSP '17)*. https://doi.org/10.1145/3132747.3132748

[16] Luke Nelson, Emina Torlak, and Xi Wang. 2021. *A proof-carrying approach to building correct and flexible in-kernel verifiers.* Retrieved October 25, 2023 from https://homes.cs.washington.edu/~lukenels/slides/2021-09-23-lpc21.pdf

[17] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the Linux kernel *(OSDI 20)*. https://www.usenix.org/conference/osdi20/presentation/nelson

[18] Valentina Palmiotti. 2021. *Kernel Pwning with eBPF - a Love Story.* Retrieved October 25, 2023 from https://chompie.rip/Blog+Posts/Kernel+Pwning+with+eBPF+-+a+Love+Story

[19] Manfred Paul. 2020. *CVE-2020-8835: Linux kernel privilege escalation via improper eBPF program verification.* Retrieved October 25, 2023 from https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification

[20] Manfred Paul. 2021. *ZDI-20-1440: An Incorrect Calculation Bug in the Linux Kernel eBPF Verifier.* Retrieved October 25, 2023 from https://www.zerodayinitiative.com/blog/2021/1/18/zdi-20-1440-an-incorrect-calculation-bug-in-the-linux-kernel-ebpf-verifier

[21] Jay Schulist, Daniel Borkmann, and Starovoitov Alexei. . *A thorough introduction to eBPF.* Retrieved October 25, 2023 from https://lwn.net/Articles/740157/

[22] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs *(ASPLOS 2006)*. https://infradive.com/docs/armando_solar_lezama/Combinatorial-Sketching-for-Finite-Programs.pdf

[23] Joe Stringer. 2018. *Document navigating BPF verifier complexity.* Retrieved October 25, 2023 from https://github.com/cilium/cilium/issues/5130

[24] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers *(CGO 2022)*. https://arxiv.org/pdf/2105.05398.pdf

[25] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification *(CGO 2023)*. https://harishankarv.github.io/assets/files/agni-cav23.pdf

[26] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure *(OSDI 2014)*. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wang_xi