EPiC Series in Computing

Volume 100, 2024, Pages 198–221

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Automatic Bit- and Memory-Precise Verification of eBPF Code

Martin Bromberger[1], Simon Schwarz[1,2], and Christoph Weidenbach[1]

[1] Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
[2] Graduate School of Computer Science, Saarbrücken, Germany

Abstract

We propose a translation from eBPF (extended Berkeley Packet Filter) code to CHC (Constrained Horn Clause sets) over the combined theory of bitvectors and arrays. eBPF is in particular used in the Linux kernel where user code is executed under kernel privileges. In order to protect the kernel, a well-known verifier statically checks the code for any harm and a number of research efforts have been performed to secure and improve the performance of the verifier. This paper is about verifying the functional properties of the eBPF code itself. Our translation procedure `bpfverify` is precise and covers almost all details of the eBPF language. Functional properties are automatically verified using `z3`. We prove termination of the procedure and show by real world eBPF code examples that full-fledged automatic verification is actually feasible.

1 Introduction

Program and system analysis have seen tremendous progress in recent years, in particular due to the increased performance of automated reasoning tools for SAT (Propositional Satisfiability) [11], SMT (Satisfiability Modulo Theories) [48], and first-order logic [1, 6]. Examples of dedicated mostly push-button verification tools are Dafny [45], SeaHorn [32], Why3 [26], and Kratos2 [30], that rely on SAT and SMT engines such as, for example, Z3 [21], CVC5 [4], and MathSAT [18]. Constrained Horn Clauses (CHC) [12] constitute a useful intermediate formal language between the actual target language and logical reasoning tools [32, 20, 27, 13]. General higher-order interactive reasoning tools such as Isabelle [49] and Coq [7] typically serve the purpose of full-fledged verification and are additionally supported by first-order logic reasoning technology [23] as, for example, performed by E [54], SPASS [60], and Vampire [42].

While many verification approaches start with highly expressible languages, we concentrate on the simple eBPF programming language [55] that is somewhat closer to the actual hardware than LLVM IR [43]. The eBPF language is in particular used to run user programs in the Linux kernel. It is a promising target for fully automatic verification because it is essentially loop-free and provides only a restricted set of instructions. In order to prevent any harm from the Linux kernel, a verifier based on static analysis checks the eBPF programs first. It is verified to be bug-free [58] but not complete, so it may reject harmless programs. There is research on improving the verifier, e.g., [28]. However, the verifier prevents the kernel only from being harmed by unauthorized memory access, crashes and resource exhaustion. It does not provide any functional correctness guarantees on the user program. To the best of our knowledge, there has not yet been a dedicated automatic bit- and memory-precise verification accounting for functional properties of eBPF programs. We translate eBPF programs with our tool `bpfverify` into CHC sets modulo the theories of bitvectors, modeling bit operations, and arrays, modeling eBPF memory. For both theories we rely on the respective SMT-LIB [5] formalization. The eBPF language does not come with a formal semantics, but we exhaustively checked our translation into CHCs against the `bpf-conformance` test suite [40] without finding any differences, see Section 5.1. Our encoding is not exhaustive for eBPF kernel-helper functions which we currently model as black boxes without any guarantees. This abstraction is not in conflict with any of our case studies but is subject to future work, see Section 3.2.4. Our translation is suitable for precisely verifying any functional property that can be expressed by a universally quantified boolean constraint over the input and output variables. Moreover, we can verify invariants of eBPF programs. For example, general invariants include the absence of integer overflows or well-formedness of memory accesses. We present two specific case studies, Section 5.2, a firewall where we can for example automatically verify properties about the acceptance and rejection of packets, and a blocking filter for system calls where we can for example automatically verify that all accepted system calls are not contained in some blocklist.

The paper is now organized as follows: after an introduction to the eBPF language and our CHC target language, Section 2, we present our translation from eBPF into CHC sets, Section 3. The translation is presented by representative examples, where the overall translation scheme is contained in the appendix that will be part of an extended version of this paper. Section 4 introduces our verification machinery on the CHC encoding and is followed by two case studies in Section 5. Finally, Section 6, discusses the obtained results and points to future work.

2 Preliminaries

We first introduce eBPF in Section 2.1. Afterwards, we introduce some background on constrained Horn clauses, Section 2.2.

2.1 The eBPF Language

Historically, eBPF (extended Berkeley Packet Filter) [55] was introduced to the Linux kernel in 2014. It allows executing user-supplied programs in the privileged kernel context. This, for example, can be used for network packet filtering directly on kernel-level. This improves performance, as network packets need not be copied to userspace. Earlier, a similar, easier approach was implemented by the "classical" BPF language, which is succeeded by eBPF.

As directly executing code in the kernel is a security risk, eBPF imposes restrictions on all programs that are run in the kernel. These restrictions are checked by the so-called *kernel verifier*. The eBPF instruction set is comparatively simple, which makes checking those restrictions feasible in practice.

eBPF code is run in a *virtual machine* in the Linux kernel. Inside the virtual machine, an eBPF program has access to eleven 64-bit registers and some restricted memory regions. Furthermore, it can call a kernel helper function. Out of the eleven registers, the registers r_0 to r_9 are general-purpose registers. The register r_{10} is a special read-only register pointing to the stack. The stack memory always has a fixed-size of 512 byte. Other memory may be statically allocated before program execution, but cannot be dynamically allocated.

The eBPF Instruction Set eBPF provides an instruction set that is run inside the virtual machine. The instruction set consists of about 100 instructions. eBPF instructions, in general[1], consist of the following 64-bit encoding:

immediate value	offset	source register	dest. register	opcode
32 bit	16 bit	4 bit	4 bit	8 bit

The `opcode` of an instruction describes the operation that should be performed. We can classify instructions into four categories based on their `opcode`:

- Arithmetic-logic operations, e.g. addition, multiplication and bitwise arithmetic.
- Control flow instructions, i.e. jumps and conditional jumps. Note that jump target locations are always static and cannot depend on register content. Only the condition for conditional jumps may depend on the current registers.
- Memory load and store instructions.
- Call instructions, which have two different contexts. First, they can call an "eBPF kernel helper function". This is the interface for an eBPF program to interact with the kernel. Second, call instructions can call another eBPF function by using tail recursion. This feature was proposed in 2017, and was subsequently added to the kernel, but is still not widely used. Our current implementation does not support this feature.

  Call instructions respect the following calling convention: The registers r_0 to r_5 are callee-saved and can be overwritten by the called function. The registers r_6 to r_9 are caller-saved.

The eBPF instruction set is an extension of the classical BPF instruction set and is almost fully backwards-compatible with BPF. Some programs are hand-written in the eBPF instruction set. However, typically an eBPF program is created by first writing a program in another programming language. This program is then checked for compatibility with the eBPF instruction set and compiled to eBPF by a dedicated compiler. For instance, LLVM [43] implements such a checker and compiler, see Example 1. Usually, programs are written in C and make use of special eBPF C libraries. Then, they are compiled to eBPF with LLVM. For a variant of eBPF for embedded systems, CertrBPF [63] provides a formally verified compiler.

Example 1. *The C program below was compiled with LLVM* `clang -c -target bpf` *to the following eBPF instructions:*

```c
long arr[] = {0, -2, -4};
long func(unsigned long x)
{
        if (x <= 2)
                return arr[x];
        return x;
}
```

```
0: r0 = r1
1: if r0 > 2 goto +5
2: r0 <<= 3
3: r1 = <dynamically loaded>
5: r1 += r0
6: r0 = *(u64 *)(r1 + 0)
7: exit
```

Loading eBPF in the Linux Kernel When loading a program into the kernel, the loader must first set up all memory. Every program automatically receives 512 bytes of stack memory. All extra memory, i.e. for global variables, is set up as a custom *eBPF map*. A map is a fixed-size memory region. Then, all instructions that use this map are filled in with the according address. In the above example, the value of instruction 3 would be replaced by a pointer to a memory region containing the array `arr`.

[1]There is a single exemption for loading a 64-bit immediate value. This load instruction is 128 bit long.

The eBPF Verifier Before executing the program in a privileged context, the kernel verifies the safety of the program. In this step, only the *absence of malicious* behavior in the program is checked. A malicious program in a privileged context could easily crash or hang the system or, even worse, modify kernel memory to gain additional privileges. To verify the absence of this behavior, the kernel *statically* verifies some properties of a given eBPF program before executing it. During runtime, no additional checks are performed.

The statically verified properties heuristically *underapproximate* the set of safe programs. It is common for the verifier to reject non-malicious programs if the absence of malicious behavior could not be proven [28]. In contrast, unsafe programs that pass the verifier pose a huge security risk, as any user can supply an eBPF program to be run in the kernel context [50].

Internally, the Linux kernel uses an abstract interpretation to perform a range analysis on the eBPF registers. The soundness of this range analysis has been formally verified [58, 9]. The verification translates the verifier source-code from the Linux kernel automatically into verification conditions, which are then checked. Furthermore, fuzzing-based approaches that test correctness of the Linux kernel eBPF verifier [47, 37] have been developed. Last, research has focused on making the eBPF verifier more precise, i.e. rejecting less safe programs. To this end, PREVAIL [28] uses a more refined abstract interpretation that accepts more programs. Still, programs that pass the PREVAIL verifier exhibit all guarantees that are provided by the standard eBPF kernel verifier.

The properties that are verified by the kernel are:

- The eBPF program must stay within an **instruction size limit**. Otherwise, by supplying exceedingly long programs, a malicious user can make the kernel hang.
- The eBPF program may **not loop indefinitely**, as this, again, hangs the kernel. In early versions of eBPF, this was handled by not allowing any backwards jumps, which guarantees termination. In newer versions, bounded loops are allowed. However, the verifier must be able to infer *simple* loop exit conditions to accept the program.
- There can be **no recursion** within an eBPF program, for the same reasons as above.
- Every **memory access** must be within an allowed region. This is checked by a heuristic range analysis of all register values. If the correctness cannot be inferred by this heuristic, the program is rejected.

A more detailed explanation of the kernel verifier can be found in the official documentation [24] or in [28]. Overall, the encoding we present in Section 3 can encode eBPF programs that do not pass the verifier. However, decidability of the resulting clauses is only guaranteed if the encoded program is accepted by the verifier. In particular, Theorem 7 only holds for eBPF programs that are accepted by the verifier.

Contexts of eBPF Programs In the Linux kernel, there are different dedicated contexts where an eBPF program can be executed. The context must be specified when loading a program into the kernel. Most prominently, the *Express Data Path (XDP)* context allows for packet filtering and modification. When loaded into the `seccomp-bpf` context, an eBPF program can filter system calls of other programs. There are more contexts for eBPF programs in the Linux kernel, such as tracing programs or controlling resource limits. Loading eBPF to some contexts such as XDP is available to non-privileged users, while other contexts can only be accessed by privileged users.

Even outside the Linux kernel, eBPF has recently gained popularity. For example, it is used as the smart-contract specification language of the Solana blockchain [62]. Our encoding to constrained Horn clauses in Section 3 is generic with respect to the context. Hence, eBPF in arbitrary contexts can be verified by our method if an appropriate specification is provided.

2.2 Constrained Horn Clauses

A constrained Horn clause is a first-order formula of the following shape:

$$\forall x_1, \dots, x_n. \quad \varphi \mid\mid B_1 \wedge \dots \wedge B_k \Rightarrow H$$

for $k \geq 0$. In such a clause, $V = \{x_1, \dots, x_n\}$ is called the set of variables appearing in the clause, φ is called the clause *constraint*, the B_i's are called *body atoms*, H is called the *head* of the clause, and $\mid\mid$ is semantically interpreted as a conjunction $\wedge$ but we use a different symbol than $\wedge$ to syntactically separate the constraint from the body atoms. A constraint Horn clause can be equivalently written in its disjunctive form: $\forall x_1, \dots, x_n. \quad \neg\varphi \vee \neg B_1 \vee \dots \vee \neg B_k \vee H$. A constraint φ is either $\top$ or a conjunction of literals constructed from the variables V, and interpreted functions and predicates from a *background theory* $\mathcal{A}$. In this paper, we always use the background theory of bitvectors in combination with arrays (`ABV`), to model machine registers and memory. Each body atom B_i has the form $P_i(\vec{y_i})$, where P_i is an uninterpreted predicate symbol and $\vec{y_i}$ is a vector of variables from V with a length equal to the arity of P_i, and corresponds to a negative literal in the disjunctive version of the clause. The head H is either $\perp$, which denotes that the disjunctive version of the clause has no positive literal, or it is an atom of the form $P(\vec{y})$, where P is an uninterpreted predicate symbol and $\vec{y}$ is a vector of variables from V with a length equal to the arity of P. Note that the predicates $P_1, \dots, P_k, P$ are not necessarily distinct. In this work, we assume all constrained Horn clauses are implicitly universally quantified and, hence, omit the quantification unless we want to highlight the variable names. A clause with $H = \perp$ is called a *query* and clauses that have no body predicates (i.e. $k = 0$) and $H \neq \perp$ are called *facts*.

Even though φ must be a conjunction by definition, in the following we allow φ to be any boolean combination of literals constructed from the variables V, and interpreted functions and predicates from a *background theory* $\mathcal{A}$. This is justified, as through CNF transformation and splitting of clauses, any such clause can be turned into a set of constraint Horn clauses where the constraints are conjunctions of literals [27]. For simplicity, we also call any such formula φ simply a *boolean $\mathcal{A}$ constraint.*

Example 2. *Consider the following set of constrained Horn clauses:*

$$N = \left\{ \begin{array}{rlll} \mathrm{bvle}(x, 100) \mid\mid P(x) & \Longrightarrow & Q(x) & \textit{(Clause)} \\ x = 42 \mid\mid & \Longrightarrow & P(x) & \textit{(Fact)} \\ \mathrm{bvge}(x, 0) \mid\mid Q(x) & \Longrightarrow & \perp & \textit{(Query)} \end{array} \right\}$$

CHC Verification The verification problem for a CHC N is asking if N is unsatisfiable, i.e. $N \models \perp$. We call such a set of constrained Horn clauses unsatisfiable or inconsistent. Contrary, if N is satisfiable, it always has a least model [46, 38, 16].

The detection of unsatisfiable clause sets can also be formulated in a deductive fashion: Any derivation of $\perp$ from a CHC is justified by a sequence of hyper-resolution [61] steps. Hence, such a derivation can be captured by Definition 3, adapted from [12].

Definition 3 (Bottom-up Derivation)**.** *A bottom-up derivation maintains a set of fact clauses of the form $\varphi \mid\mid \ \Rightarrow P(\vec{y})$. It then applies hyper-resolution on clauses $\psi \mid\mid B_1 \wedge \ldots \wedge B_k \Rightarrow H$,* *resolving away all body atoms $B_1, \ldots, B_k$ using fact clauses. The clauses are inconsistent if it derives a clause of the form $\forall x_1, \ldots, x_n.\varphi \mid\mid \Rightarrow \bot$ such that $\exists x_1, \ldots, x_n.\varphi$ is satisfied.*

Note that for this derivation method (and many other methods for CHCs) a theory solver for the quantifier-free fragment of the background theory is sufficient for background theory reasoning.

Example 4 (Bottom-up Derivation)**.** *Considering the clause set from Example 2, a bottom-up derivation starts with the following resolution step:*

$$\left\{ \begin{array}{ll} x = 42 & \mid\mid \qquad \Longrightarrow P(x) \\ \mathrm{bvle}(x, 100) & \mid\mid P(x) \Longrightarrow Q(x) \end{array} \right\} \rightsquigarrow x = 42 \wedge \mathrm{bvle}(x, 100) \mid\mid \Longrightarrow Q(x)$$

The next resolution step happens between the newly established fact and the query:

$$\left\{ \begin{array}{ll} x = 42 \wedge \mathrm{bvle}(x, 100) & \mid\mid \qquad \Longrightarrow Q(x) \\ \mathrm{bvge}(x, 0) & \mid\mid Q(x) \Longrightarrow \bot \end{array} \right\} \rightsquigarrow x = 42 \wedge \mathrm{bvle}(x, 100) \wedge \mathrm{bvge}(x, 0) \mid\mid \Longrightarrow \bot$$

As the background conjunction $x = 42 \wedge \mathrm{bvle}(x, 100) \wedge \mathrm{bvge}(x, 0)$ is satisfiable, the overall set N is unsatisfiable.

Constrained Horn in Practice In practice, checking CHCs for satisfiability can be done by various tools, such as SPACER [31], Ultimate TreeAutomizer [25], and ELDARICA [35]. There are many application areas for CHC solvers, for instance model checking [36, 8], verification of inductive invariants [32], verification of distributed and parameterized systems [29, 51, 33], and type inference [56, 57]. Moreover, Horn clauses naturally encode the set of reachable states of sequential programs, and hence, have been used for this purpose for multiple languages, e.g. by SeaHorn for C and LLVM [32].

In this paper, we have chosen to target CHC. For CHC, there exist multiple solvers directly supporting the combined theories of bitvectors and arrays, which we need for our encoding. Furthermore, CHCs are a natural choice for encoding programs with loops. For loop-free programs, bounded model checking formats such as AIGER [10] would be another choice. Our eventual target is extending the encoding with (terminating) eBPF loops, see Section 6 and the discussion after Theorem 7 in Section 4.

Other approaches in bounded model checking include SeaBMC [52], which translates programs into a custom intermediate representation, and SMACK [53], which uses the Boogie [44] intermediate verification language. In this work, we choose CHC in the SMT-LIB format as our intermediate language, as it is a widely used format: First, our encoded CHC can be input to many existing CHC solvers. Moreover, this format can also be read by some bounded model checker, e.g. the `bmc` engine in `z3` (see Section 5).

3 Encoding eBPF to CHC

In this section, we present our encoding of the eBPF operational semantics into sets of CHCs. Let P be an eBPF program. We introduce a function encode(P) that returns a set of CHCs.

Notationally, we will refer to functions and types with their names in SMT-LIB. For readability, we use standard prefix notation for function application, and infix notation for equality. This section contains only some exemplary encodings. All encoding rules are given in SMT-LIB syntax in the appendices A–C.

3.1 Program States in CHC

We will model a program state of an eBPF program as an *atom* in the CHC fragment. An atom capturing a program state looks as follows:

$$L_{pc}(r_0, r_1, \ldots, r_{10}, M)$$

Recall from Section 2.1 that a state of the eBPF virtual machine consists of the following parts:

- A program counter pc that saves the current instruction.
- The eleven 64-bit registers $r_0, r_1, \ldots, r_{10}$. Those registers are of type `BitVec 64`.
- The current memory M in this execution step. As the memory is indexed by 64-bit pointers and stores single bytes, its type is `Array ((BitVec 64) (BitVec 8))`.

3.2 Encoding eBPF Instructions as Clauses

A single eBPF instruction corresponds to a constrained Horn clause in our encoding. In general, such a program transition will be encoded as a clause

$$\varphi \mid\mid L_{pc}(r_0, \ldots, r_{10}, M) \implies L_{pc'}(r'_0, \ldots, r'_{10}, M')$$

In this section, for each instruction opcode, we give an encoding of the respective instruction to a constrained Horn clause. Note that previously, in the context of eBPF verification, there was *no formal definition* of the full operational semantics for eBPF to the best of our knowledge. In practice, the eBPF semantics are defined by the implementation of the virtual machine in the Linux kernel, which is treated as authoritative. With our encoding, we encode the full operational semantics of eBPF in CHC, hence giving eBPF a formal semantics. To show the adequacy of our defined operational semantics, we rely on extensive testing. In Section 5.1, we describe our tests for ensuring equivalence between our defined operational semantics and the real-world behavior of the Linux kernel.

3.2.1 Encoding of Arithmetic-Logic Operations

Arithmetic-logic instructions increase the program counter by one and perform an operation on the destination register `dst`. There is a difference between instructions that use an *immediate value* (`imm`) and instructions where both operands are register values. For example the instruction `r0 += 42` uses the immediate value 42, which is encoded in the instruction. In contrast, the instruction `r0 += r1` uses the content of register `r1` as an operand.

In CHC, the actual arithmetic computation is encoded in a background formula using SMT operations on the corresponding bitvectors. The encoding of some representative arithmetic-logic instructions is shown in Table 1. Note that the encodings are very similar, with only the corresponding background function differing. A concise representation of all arithmetic-logic instructions and their corresponding encoding can be found in Appendix A. A special case includes division by zero. Note that the kernel verifier does not track precise register values for most registers. Hence, it cannot reliably detect divisions by zero. To avoid crashes, eBPF defines division by zero to result in a zero value.

Instr. at $pc = i$	CHC added by encode(P)
`r0 += imm`	$r_0' = \text{bvadd}(r_0, \text{imm}) \mathrel{\|\|} L_i(r_0, r_1, \ldots, r_{10}, M) \implies L_{i+1}(r_0', r_1, \ldots, r_{10}, M)$
`r0 += r1`	$r_0' = \text{bvadd}(r_0, r_1) \mathrel{\|\|} L_i(r_0, r_1, \ldots, r_{10}, M) \implies L_{i+1}(r_0', r_1, \ldots, r_{10}, M)$
`r0 -= imm`	$r_0' = \text{bvsub}(r_0, \text{imm}) \mathrel{\|\|} L_i(r_0, r_1, \ldots, r_{10}, M) \implies L_{i+1}(r_0', r_1, \ldots, r_{10}, M)$
`r0 -= r1`	$r_0' = \text{bvsub}(r_0, r_1) \mathrel{\|\|} L_i(r_0, r_1, \ldots, r_{10}, M) \implies L_{i+1}(r_0', r_1, \ldots, r_{10}, M)$
`r0 |= imm`	$r_0' = \text{bvor}(r_0, \text{imm}) \mathrel{\|\|} L_i(r_0, r_1, \ldots, r_{10}, M) \implies L_{i+1}(r_0', r_1, \ldots, r_{10}, M)$
`r0 |= r1`	$r_0' = \text{bvor}(r_0, r_1) \mathrel{\|\|} L_i(r_0, r_1, \ldots, r_{10}, M) \implies L_{i+1}(r_0', r_1, \ldots, r_{10}, M)$
`r0 /= imm`	$r_0' = \begin{cases} 0 & \text{if imm} = 0 \\ \text{bvudiv}(r_0, \text{imm}) & \text{otherwise} \end{cases} \mathrel{\|\|} L_i(r_0, r_1, \ldots, r_{10}, M) \implies \ldots$
$\vdots$	$\vdots$

Table 1: Encoding of arithmetic-logic instructions

3.2.2 Encoding of Control Flow Operations

In a control flow instruction, the program counter is (conditionally) not only increased by one, but by a *fixed* offset, named off. In a conditional control flow instruction, the encoding makes use of *two* separate constrained Horn clauses: One clause for the case that the condition is satisfied, and one clause for the complement. The condition of the jump is always part of the background constraints. Table 2 shows the encoding of some exemplary control-flow instructions. A full reference is given in Appendix B.

Instruction at $pc = i$	CHC added by encode(P)
`PC += off`	$\top \mathrel{\|\|} L_i(r_0, \ldots, r_{10}, M) \implies L_{i+\text{off}}(r_0, \ldots, r_{10}, M)$
`PC += off if r0==imm`	$r_0 = \text{imm} \mathrel{\|\|} L_i(r_0, \ldots, r_{10}, M) \implies L_{i+\text{off}}(r_0, \ldots, r_{10}, M)$ $r_0 \neq \text{imm} \mathrel{\|\|} L_i(r_0, \ldots, r_{10}, M) \implies L_{i+1}(r_0, \ldots, r_{10}, M)$
`PC += off if r0==r1`	$r_0 = r_1 \mathrel{\|\|} L_i(r_0, \ldots, r_{10}, M) \implies L_{i+\text{off}}(r_0, \ldots, r_{10}, M)$ $r_0 \neq r_1 \mathrel{\|\|} L_i(r_0, \ldots, r_{10}, M) \implies L_{i+1}(r_0, \ldots, r_{10}, M)$
`PC += off if r0 < imm`	$\text{bvult}(r_0, \text{imm}) \mathrel{\|\|} L_i(r_0, \ldots, r_{10}, M) \implies L_{i+\text{off}}(r_0, \ldots, r_{10}, M)$ $\neg\,\text{bvult}(r_0, \text{imm}) \mathrel{\|\|} L_i(r_0, \ldots, r_{10}, M) \implies L_{i+1}(r_0, \ldots, r_{10}, M)$
`PC += off if r0 < r1`	$\text{bvult}(r_0, r_1) \mathrel{\|\|} L_i(r_0, \ldots, r_{10}, M) \implies L_{i+\text{off}}(r_0, \ldots, r_{10}, M)$ $\neg\,\text{bvult}(r_0, r_1) \mathrel{\|\|} L_i(r_0, \ldots, r_{10}, M) \implies L_{i+1}(r_0, \ldots, r_{10}, M)$
$\vdots$	$\vdots$

Table 2: Encoding of control flow instructions

3.2.3 Encoding of Memory Usage

Memory accesses are handled by the background SMT theory `Array`. The underlying memory is partitioned into single bytes. However, there are single instructions that load or store multiple bytes directly. For those instructions, the final value is built by using the `extract` or `concat` functions of the bitvector theory. Overall, there are four load instructions (1 byte load, 2 byte load, 4 byte load, 8 byte load). Furthermore, there are four respective store instructions that write register contents to memory. In addition, there is also an immediate version for all store

instructions.
Example load instructions are depicted in Table 3, some exemplary store instructions in Table 4. A full reference of all memory instructions can be found in Appendix C. Syntactically, we use the representation $M[x]$ for reading array access. In SMT-LIB, this corresponds to the function `(select M x)`.

Instruction if $pc = i$	CHC added by encode(P)
`r0 = *(u8*) *(r1 + off)`	$r_0' = M[\text{bvadd}(r_1, \text{off})] \mid\mid L_i(r_0, \dots, r_{10}, M) \implies L_{i+1}(r_0', \dots, r_{10}, M)$
`r0 = *(u16*) *(r1 + off)`	$x_1 = M[\text{bvadd}(r_1, \text{off})] \wedge x_2 = M[\text{bvadd}(r_1, \text{off}{+}1)] \wedge r_0' = \text{concat}(x_2, x_1) \mid\mid L_i(r_0, r_1, \dots, r_{10}, M) \implies L_{i+1}(r_0', r_1, \dots, r_{10}, M)$
⋮	⋮

Table 3: Encoding of memory load instructions

Instruction if $pc = i$	CHC added by encode(P)
`*(u8*) *(r0 + off) = r1`	$M' = \text{store}(M, \text{bvadd}(r_0, \text{off}), \text{extract}(7, 0, r_1)) \mid\mid L_i(r_0, \dots, r_{10}, M) \implies L_{i+1}(r_0, r_1, \dots, r_{10}, M')$
`*(u16*) *(r0 + off) = r1`	$M' = \text{store}(M, \text{bvadd}(r_0, \text{off}), \text{extract}(7, 0, r_1)) \wedge M'' = \text{store}(M', \text{bvadd}(r_0, \text{off}{+}1), \text{extract}(15, 8, r_1)) \mid\mid L_i(r_0, \dots, r_{10}, M) \implies L_{i+1}(r_0, r_1, \dots, r_{10}, M'')$
⋮	⋮

Table 4: Encoding of memory store instructions

3.2.4 Encoding of Call Instructions

There are two types of call instructions. First, there are call instructions that call into other eBPF functions that are declared in the same program. This option was introduced in 2017 and is not widely used. Hence, we currently do not provide an encoding for such calls. Second, there are "kernel-helper" functions that can be called. These provide an interface to the kernel. For example, kernel helper functions are:

- `bpf_get_prandom_u32`, which returns a random integer.
- `bpf_trace_printk`, a printf-like construct for debugging.
- `bpf_map_lookup_elem`, `bpf_map_update_elem` provide a map data structure for our eBPF program.

Currently, the encoding of a call instruction makes *almost no assumptions* about the called function. Instead, according to the calling convention, it sets $r_0 - r_5$ to an arbitrary (undefined) value, while keeping the other registers. This is justified by the eBPF calling convention, as $r_0 - r_5$ are caller-saved registers, $r_6 - r_9$ are callee-saved, and r_{10} is read-only. Furthermore, it assumes that all memory that is available to the eBPF program is unmodified. This is true for almost all kernel helpers, except some (rarely used) helpers that modify packets, i.e. by recalculating a checksum.

This encoding models all kernel helper functions as non-deterministic functions that leave memory untouched. This is accurate for some kernel functions such as `bpf_get_prandom_u32`. However, some functions provide more guarantees. In particular, this applies to the “map” functionality (`bpf_map_lookup_elem`, `bpf_map_update_elem`), which basically implement a key-value map data structure, and is commonly used. Similarly, other functions provide other guarantees as well, as can be seen in Example 5. Implementing a more precise model for kernel functions is subject to further work, see Section 6.

Example 5 (Incompleteness with Kernel Helpers)**.** *Consider the following function fragment:*

```
int x = bpf_get_smp_processor_id();
int y = bpf_get_smp_processor_id();
return x == y;
```

In practice, this code always returns 1*. However, this is not captured with our current kernel helper encoding.*

3.3 Encoding Program Flow

Every eBPF program has a specified entrypoint. If this entrypoint is located at instruction i, we establish the following additional rule in encode(P):

$$\top \mid\mid \mathrm{Loader}(r_0, \dots, r_{10}, M) \implies L_i(r_0, \dots, r_{10}, M)$$

Furthermore, every eBPF program has one or multiple `exit` statements. We establish a Final predicate with the following rule:

Instruction if $pc = i$	CHC added by encode(P)
`exit`	$\top \mid\mid L_i(r_0, r_1, \dots, r_{10}, M) \implies \mathrm{Final}(r_0, r_1, \dots, r_{10}, M)$

3.4 Encoding Dynamic Loading

In eBPF, the stack, global variables, extra memory (eBPF maps), and the input to the program is stored in dynamically loaded sections. In the compiled program code, an instruction that is loading a pointer to such a section will receive a placeholder value. The actual non-deterministic pointer value is then filled in at loading time.

A single section is always a continuous region in memory. In theory, no guarantees about the ordering and placement of different sections are made. However, in practice, all loaders that we are aware of follow a sequential loading pattern, i.e. placing sections in their defined order next to each other. Furthermore, eBPF does not allow pointer comparison in all non-privileged eBPF contexts. Hence, while in theory the actual pointer address is non-deterministic, it is sufficient for our model to model them as fixed, deterministic addresses. This significantly simplifies the encoding of dynamic loading.

In `bpfverify`, a single base address c = `0x60000000` is fixed. Furthermore, a value k is chosen bigger than the maximum size of any section. Then, the n-th section is loaded at address $c + n \cdot k$. In CHC, we use an additional Loader$(r_0, \dots, r_{10}, M)$ atom, and add the following clause to encode(P):

$$M' = \mathrm{load}(M, [s_1, s_2, \dots, s_n]) \mid\mid \mathrm{Initial}(r_0, \dots, r_{10}, M) \implies \mathrm{Loader}(r_0, \dots, r_{10}, M')$$

Where the macro load$(M, [s_1, s_2, \ldots, s_n])$ recursively stores all bytes of all sections s_i into the memory, i.e. load$(M, [s_1, s_2, \ldots, s_n]) = \text{store}(\text{store}(\text{store}(\ldots, (c+k), s_1^0), (c+1), s_0^1), (c+0), s_0^0)$

Example 6. *Recall the program P from Example 1. The eBPF instructions are printed on the left. The set of resulting CHC clauses* encode(P) *is displayed on the right. In this example, the loading base address was* `0x6000`*.*

```
0: r0 = r1
1: if r0 > 2 goto +5

2: r0 <<= 3
3: r1 = <dyn. loaded>
5: r1 += r0
6: r0 = *(u64 *)(r1 + 0)
7: exit
```

$$\left\{\begin{array}{rcl}
\top \;\|\; \text{Initial}(r_0,\ldots,r_{10},M) & \Longrightarrow & \text{Loader}(r_0,\ldots,r_{10},M) \\
\begin{array}{l} M_1 = \text{store}(M, \mathit{0x6000}, 0) \\ M_2 = \text{store}(M_1, \mathit{0x6001}, -2) \\ M_3 = \text{store}(M_2, \mathit{0x6002}, -4) \end{array} \;\|\; \text{Loader}(r_0,\ldots,r_{10},M) & \Longrightarrow & L_0(r_0,\ldots,r_{10},M_3) \\
\top \;\|\; L_0(r_0,r_1,\ldots,r_{10},M) & \Longrightarrow & L_1(r_1,r_1,\ldots,r_{10},M) \\
\text{bvgt}(r_0,2) \;\|\; L_1(r_0,r_1,\ldots,r_{10},M) & \Longrightarrow & L_6(r_0,r_1,\ldots,r_{10},M) \\
\neg\,\text{bvgt}(r_0,2) \;\|\; L_1(r_0,r_1,\ldots,r_{10},M) & \Longrightarrow & L_2(r_0,r_1,\ldots,r_{10},M) \\
r_0' = r_0 \ll 3 \;\|\; L_2(r_0,r_1,\ldots,r_{10},M) & \Longrightarrow & L_3(r_0',r_1,\ldots,r_{10},M) \\
r_1' = \mathit{0x6000} \;\|\; L_3(r_0,r_1,\ldots,r_{10},M) & \Longrightarrow & L_4(r_0,r_1',\ldots,r_{10},M) \\
r_1' = \text{bvadd}(r_0,r_1) \;\|\; L_4(r_0,r_1,\ldots,r_{10},M) & \Longrightarrow & L_5(r_0,r_1',\ldots,r_{10},M) \\
r_0' = \text{select}(M,r_1) \;\|\; L_5(r_0,r_1,\ldots,r_{10},M) & \Longrightarrow & L_6(r_0',r_1,\ldots,r_{10},M) \\
\top \;\|\; L_6(r_0,r_1,\ldots,r_{10},M) & \Longrightarrow & \text{Final}(r_0,r_1,\ldots,r_{10},M)
\end{array}\right\}$$

Our encoding function encode(P) always produces a CHC set that does not yet contain any query or fact clauses. The set encode(P) always contains a rule that has the initial atom Initial$(r_0, \ldots, r_{10}, M)$ as body atom, but no rule that has Initial$(r_0, \ldots, r_{10}, M)$ as head. Similarly, Final$(r_0, \ldots, r_{10}, M)$ only occurs in the head of rules, and not in the body. To prove properties about P, we will establish queries and facts involving the initial and final atoms in the next section.

4 Verification of eBPF Programs

In this section, we present the general framework for proving eBPF properties on the basis of CHCs generated by the encoding defined in Section 3. Firstly, we recall that the CHC fragment for eBPF is decidable. Recall that a Horn clause set is *recursive* if for some predicate P there exists a cycle in its predicate dependency graph, i.e., the directed graph that has a node for every occurring predicate and a directed edge $P \to Q$ for every pair (P, Q) of predicates such that P appears in a body atom and Q in the head atom of the same clause.

Theorem 7 (Decidability of Non-Recursive CHCs over Bitvectors and Arrays)**.** *Let N be a CHC over the combined theory of bitvectors and arrays (*`ABV`*) without recursion. Then satisfiability of N is decidable.*

Proof. By definition, see Section 2.2, the body and head atoms of all clauses in N only contain variables. Therefore, N is sufficiently complete and hierarchic superposition [3, 14, 13] is refutationally complete on N. A superposition strategy where in all clauses with body literals at least one literal is selected will terminate, because N is not recursive. Now N is satisfiable iff no clause $\forall x_1, \ldots, x_n.\phi \;\|\!\Rightarrow \bot$ is derived by this superposition strategy such that $\exists x_1, \ldots, x_n.\phi$ is satisfiable. Satisfiability of $\exists x_1, \ldots, x_n.\phi$ for `ABV` is equivalent to a `QF_ABV` problem and decidable [39]. □

Decidability of the CHCs generated by encode(P) could also be shown by bottom-up CHC evaluation [12]. We presented a proof based on hierarchic superposition (and hence also SCL [15]), because this framework provides a lot of flexibility in reasoning about a clause set. Although we did not explore this in this paper, this flexibility can result in an exponential increase in performance [17] as soon as bounded loops are considered. Furthermore, the CHC encoding supports the analysis of loop structures via established research in first-order reasoning. eBPF loops correspond to recursive predicates in the CHC encoding. There are decidability results known for first-order fragments with recursive predicates [2, 59] as well as techniques to analyze recursive predicate structures with respect to termination [34, 41].

Corollary 8 (eBPF Program Verification is Decidable)**.** *Let P be an eBPF program, let ϕ, ψ be boolean* `ABV` *constraints, and let*

$$N = \text{encode}(P) \cup \left\{ \begin{array}{ccc} \phi & \| & \implies \text{Initial}(r_0, r_1, \ldots, M) \\ \psi & \| & \text{Final}(r_0, r_1, \ldots, M) \implies \bot \end{array} \right\}$$

Then satisfiability of N is decidable.

Proof. N is not recursive and also meets all other prerequisites of Theorem 7. □

Now Corollary 8 can be used to automatically verify purely universal and purely existential properties of a program P where ϕ, ψ are boolean `ABV` constraints. Assume we want to prove $\exists \vec{x}.\{\phi\}P\{\psi\}$ where ϕ is the precondition for P and ψ its postcondition, and $\vec{x}$ binds all free variables in ϕ and ψ. Then this property holds iff

$$N = \text{encode}(P) \cup \left\{ \begin{array}{ccc} \phi & \| & \implies \text{Initial}(r_0, r_1, \ldots, M) \\ \psi & \| & \text{Final}(r_0, r_1, \ldots, M) \implies \bot \end{array} \right\}$$

is unsatisfiable. On the other hand, if we want to prove $\forall \vec{x}.\{\phi\}P\{\psi\}$ then this property holds iff

$$N = \text{encode}(P) \cup \left\{ \begin{array}{ccc} \phi & \| & \implies \text{Initial}(r_0, r_1, \ldots, M) \\ \neg\psi & \| & \text{Final}(r_0, r_1, \ldots, M) \implies \bot \end{array} \right\}$$

is satisfiable. Properties requiring quantifier alternations are beyond the scope of this paper and are also beyond existing decision procedures for `ABV`, in general.

Example 9. *Consider the program P and its encoding* encode(P) *from Example 6. We can now prove the following universal property: “If the input $x \geq 3$, then func(x) > 0”. Recall that x is passed as first parameter in r_1 by the eBPF calling convention. The result* func(x) *is returned in r_0. We construct the following clause set*

$$N = \text{encode}(P) \cup \left\{ \begin{array}{ccc} \text{bvge}(r_1, 3) & \| & \implies \text{Initial}(r_0, r_1, \ldots, M) \\ \neg\,\text{bvgt}(r_0, 0) & \| & \text{Final}(r_0, r_1, \ldots, M) \implies \bot \end{array} \right\}$$

By Corollary 8, the satisfiability of N is decidable. Here, notice that $N \not\models \bot$. Hence, the above property holds for the program.

Functional Verification With our encoding so far, we are able to prove statements about pre- and postconditions of the program. However, we cannot yet prove functional correctness properties such as “If $x \geq 3$, then P(x) = x” that relate the input to the output of a program. We will introduce a slightly modified encoding to be able to capture such properties in the next section while preserving Theorem 7 and Corollary 8. We modify the atoms representing program states in our CHC encoding to:

$$L_{pc}(r_0, \ldots, r_{10}, M, r_0^0, \ldots, r_{10}^0, M^0)$$

Now, each atom in addition captures the *original input* values in the additional variables $r_0^0, \ldots, r_{10}^0, M^0$. Those variables can be added to the Loader and Final predicate similarly. We can now adapt all of the rules from Section 3 in the following fashion:

$$\varphi \mid\mid \text{Initial}(r_0, \ldots, r_{10}, M) \implies \text{Loader}(r_0, \ldots, r_{10}, M, r_0, \ldots, r_{10}, M)$$

The first rule now simply duplicates the input values. All further rules will now only operate on the first copy, leaving the variables $r_0^0, \ldots, r_{10}^0, M^0$ completely untouched.

$$\varphi \mid\mid L_i(r_0, \ldots, r_{10}, M, r_0^0, \ldots, r_{10}^0, M^0) \implies L_{i'}(r'_0, \ldots, r'_{10}, M', r_0^0, \ldots, r_{10}^0, M^0)$$

Finally, this construction allows us to relate all input values to the values in the final state. Consider a query of the following shape:

$$\varphi \mid\mid \text{Final}(r_0, \ldots, r_{10}, M, r_0^0, \ldots, r_{10}^0, M^0) \implies \bot$$

Here, the background formula φ can range over the variables $r_0, \ldots, r_{10}, M$ as well as over the variables $r_0^0, \ldots, r_{10}^0, M^0$. The former correspond to the final state, while the latter correspond to the initial state. Hence, φ can relate input and output variables.

Example 10 (Functional Verification)**.** *Recall the program P from Example 6. We now want to prove the property "if $x \geq 3$, then func(x) $= x$". Again, x is passed in r_1 and* func(x) *is returned in r_0. Let N be the modified encoding of P. This property can be proven by considering*

$$N' = N \cup \left\{ \begin{array}{lll} \text{bvge}(r_1, 3) & \mid\mid & \implies \text{Initial}(r_0, r_1, \ldots, M) \\ r_0 \neq r_1^0 & \mid\mid & \text{Final}(r_0, r_1, \ldots, M, r_0^0, r_1^0, \ldots, M^0) \implies \bot \end{array} \right\}$$

If $N' \not\models \bot$, in all executions it must hold that $r_0 = r_1^0$, i.e., the final return value is equal to the initial first argument. Following Corollary 8, this property is decidable.

Invariant Verification Last, we give some more concrete examples for interesting verification conditions. In our encoding, we can verify invariants by adding a rule for *unsafe* states that violate our invariant:

$$\varphi \mid\mid L_{pc}(r_0, r_1, \ldots, r_{10}, M) \implies \bot \qquad\qquad \text{if } L_{pc}(r_0, r_1, \ldots, r_{10}, M) \text{ is unsafe}$$

One prominent example of such an invariant is the absence of (signed) integer overflows. Within our encoding, we need to add the queries specified in Table 5. The encoded program has no signed integer overflow if and only if the corresponding clause set is satisfiable. Similar queries can be established for unsigned integer overflows.
Similarly, we can encode invariant conditions for unsafe memory access. Table 6 gives the rules that are needed for verifying the absence of malformed memory accesses. Note that the eBPF verifier already statically checks memory accesses. However, the verifier underapproximates the set of safe programs, while we can precisely characterize the set of programs that do not employ unsafe memory accesses.

5 Evaluation

The proposed translation is currently implemented in Python with the `z3` Python bindings. The implementation takes an eBPF ELF file, and can either produce SMT-LIB output, or query the

Inst. if $pc = i$	Generated CHC
`r0 += imm`	$r_0 >_s 0 \wedge \text{imm} >_s 0 \wedge \text{bvadd}(r_0, \text{imm}) <_s 0 \mid\mid L_i(r_0, \ldots, r_{10}, M) \implies \bot$ $r_0 <_s 0 \wedge \text{imm} <_s 0 \wedge \text{bvadd}(r_0, \text{imm}) >_s 0 \mid\mid L_i(r_0, \ldots, r_{10}, M) \implies \bot$
`r0 -= imm`	$r_0 >_s 0 \wedge \text{imm} <_s 0 \wedge \text{bvadd}(r_0, \text{imm}) <_s 0 \mid\mid L_i(r_0, \ldots, r_{10}, M) \implies \bot$ $r_0 <_s 0 \wedge \text{imm} >_s 0 \wedge \text{bvadd}(r_0, \text{imm}) >_s 0 \mid\mid L_i(r_0, \ldots, r_{10}, M) \implies \bot$
`r0 *= imm`	$\neg\, \text{bvmul\_noofl}(r_0, \text{imm}) \mid\mid L_i(r_0, \ldots, r_{10}, M) \implies \bot$
`r0 = -r0`	$r_0 = -2^{63} \mid\mid L_i(r_0, \ldots, r_{10}, M) \implies \bot$
$\vdots$	$\vdots$

Table 5: Queries for checking for signed integer overflows

Instruction if $pc = i$	Generated CHC
`r0 = *(uint8_t *)(r1 + off)`	$\neg\, \text{valid}(\text{bvadd}(r_1, \text{off})) \mid\mid L_{pc}(r_0, \ldots, r_{10}, M) \implies \bot$
`*(uint8_t *)(r1 + off) = r0`	$\neg\, \text{valid}(\text{bvadd}(r_1, \text{off})) \mid\mid L_{pc}(r_0, \ldots, r_{10}, M) \implies \bot$
$\vdots$	$\vdots$

Where valid(x) is a macro that is true iff x is a pointer into any allocated memory section, i.e.
$\text{valid}(x) = \bigvee_{(s,e)\in R}(\text{bvge}(x, s) \wedge \text{bvle}(x, e))$
where R is the set containing tuples of start and end addresses of all memory section.

Table 6: Queries for checking for invalid memory accesses

solvers `z3` or `ELDARICA` directly. As the default solver, the Fixedpoint solver of `z3` with engine `tab` is selected. We have observed that, on our evaluation, this combination performs best. In particular, it is more efficient than the bounded-model checking (`bmc`) engine of `z3` on almost all tests. `z3` with engine `tab` is used in the evaluation below.

The implemented program, `bpfverify`, is available for download at the following address: `https://nextcloud.mpi-klsb.mpg.de/index.php/s/kYa2YnCGed7DkRC`.

5.1 Adequacy of the Defined Operational Semantics

In this work, we have given the first full definition of the operational semantics of eBPF. We claim that our defined operational semantics of eBPF align with the real-world semantics of eBPF, except for the differences when calling kernel helpers (Section 3.2.4).

We experimentally verified consistency between the current Linux eBPF virtual machine and our operational semantics. To this end, we use the extensive `bpf-conformance` test suite [40]. For a fixed test case, a reference result R is obtained by running the program in the Linux kernel. We then prove the following two properties with our operational semantics:

- A program run that produces the result R exists.
- No program run that produces a result different from R exists.

The `bpf-conformance` test suite consists of 148 tests. We need to exclude two tests, one which uses bounded loops and one which uses function calling. On all other tests, both properties can be verified. Overall, it takes about 20s for proving all $2 \cdot 146$ properties.

5.2 Verification of eBPF Programs

In this section, we focus on functionally verifying real-world eBPF programs. To this end, we consider two case studies where eBPF is loaded in two different contexts.

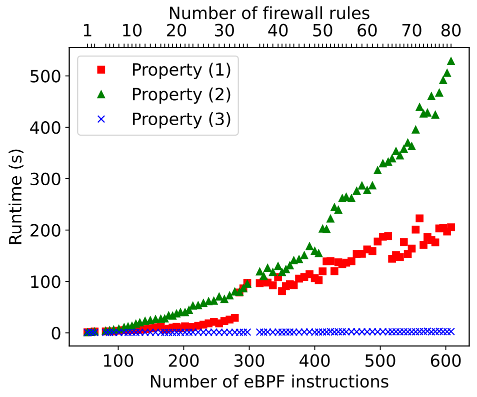
Case Study: XDP-Firewall `bpfverify` can verify properties on a firewall build in the *Express Data Path (XDP)* context. The open-source project `XDPFirewall` [22] implements a configurable firewall in XDP. In this implementation, a sequence of firewall rules is applied to network packets. A single rule can match a class of packets, and either accept or drop all packets from this class. The amount of rules is configurable, but limited to 80. The program uses kernel helper calls to implement a dynamic blacklisting and statistics functionality. As we currently do not model kernel helper calls, we disable this feature for verification.

For example, we can verify the following properties on `XDPFirewall`:

(1) There exist packets fulfilling a property φ that are accepted. For example, there exists a host in subnet `192.168.XX.XX` from which incoming packets are accepted.

(2) For all incoming packets fulfilling a property φ, a specific action is taken. For example, all packets that are not coming from a specific subnet, i.e. `192.168.XX.XX` are dropped.

(3) There cannot be an integer overflow in any execution of our firewall code.



The right figure shows the verification runtime against the number of instructions in our firewall. For evaluation, we incrementally add rules that match additional classes of packets. If more firewall rules are provided, the resulting eBPF program grows in size. Each additional firewall rule adds about 6 eBPF instructions. However, the exact amount varies due to compiler optimizations. The first 50 eBPF instructions load and check well-formedness of the corresponding network packet, and are always present even if only a single rule is loaded. For the first 50 instructions, all three properties are comparatively easy to verify. Furthermore, the firewall program only contains basic arithmetic instructions. Hence, property (3) is easily verified even for large programs. Properties (1) and (2) are also verified within few minutes for firewalls with up to 80 rules.

Case Study: `seccomp-bpf`-Filter BPF can also be used in the `seccomp-bpf` filtering context. A `seccomp-bpf` filter can restrict system calls that can be performed by a user program. For such filters, only "classical" BPF (not eBPF) can be used. However, as eBPF is backwards-compatible, our verification encoding still applies. We verify the `seccomp-bpf` filter generated by Firejail [19]. Firejail is a tool used for process isolation. In the default Firejail configuration, 70 system calls from the blocklist `@default_nodebuggers` are filtered for isolated processes. Given a blocklist, Firejail then generates a BPF program that implements filtering. This BPF program is then loaded to the kernel in the `seccomp-bpf` context. With the default blocklist, the generated BPF program has 148 instructions. We can, for example, prove the following properties about this program:

- A specific system call is blocked by the eBPF program. For example, by default Firejail blocks the `ptrace` system call which prevents debugging (0.21s verification time).
- All system calls on our blocklist are filtered (0.30s verification time).
- All other valid system calls that are not part of the blocklist are allowed (8.10s verification time).
- There cannot be an integer overflow in any execution (0.02s verification time).

6 Conclusion

We have presented a sound encoding of eBPF programs into CHCs implemented by our tool `bpfverify`. Using `z3`, we are able to verify bit- and memory-precise functional properties of real-world eBPF programs in reasonable time. Future work may be along several lines.

The eBPF language is pretty close to languages used in smart-contract design by, e.g., Solana [62] where functional properties of contracts play an important role as well. We might investigate to adjust our verification pipeline to this end.

The eBPF language is comparatively easy to verify due to its restricted instruction set. However, eBPF that is loaded into the kernel is even more restricted by the kernel verifier (see Section 2.1). In particular, the kernel verifier statically tracks register types (i.e., pointer value or scalar value) as well as possible ranges. We suppose that if this information is supplied to the eventual reasoning engine, a significant speed-up can be gained. For example, most programs do not use dynamic memory access. If the Linux kernel verifier can infer a load from a static memory access, we can simplify the load in the encoding to an array access at a constant position. Incorporating the Linux kernel verifier is quite challenging, as its current version is tightly integrated with the Linux kernel. Hence, some modifications need to be made to the verifier code to compile it without kernel dependencies.

Currently, we have only limited support for kernel-helper functions, see Section 3.2.4. This was sufficient for our two case studies but may need to be extended for further case studies. For example, the "map" functionality (`bpf_map_lookup_elem`, `bpf_map_update_elem`), which basically implements a key-value map data structure that is commonly used is a good candidate.

In newer kernel versions, bounded loops as well as tail recursion are allowed in eBPF. As every execution remains bounded, every property stays decidable. Still, the encoding needs to be modified, as currently a state is uniquely identified by its program counter, which no longer works with bounded loops. In addition to unrolling such loops and applying our current verification pipeline, we may investigate adapting superposition or SCL-based solving to terminate on such loops without unrolling, see also our discussion after the proof of Theorem 7. Ultimately, a goal would be to define a restricted eBPF language including loops where termination is guaranteed and that goes beyond a priori bounded loops. By application of superposition or SCL-based solving methods we are also no longer bound to Horn clauses and may therefore also consider query or language extensions in this direction.

Acknowledgements: We thank our anonymous reviewers for their constructive comments.

References

[1] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994. Revised version of Max-Planck-Institut für Informatik technical report, MPI-I-91-208, 1991.

[2] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Superposition with simplification as a decision procedure for the monadic class with equality. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Computational Logic and Proof Theory, Third Kurt Gödel Colloquium*, volume 713 of *LNCS*, pages 83–96. Springer, August 1993.

[3] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing, AAECC*, 5(3/4):193–212, 1994.

[4] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[6] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirovic. Superposition for higher-order logic. *J. Autom. Reason.*, 67(1):10, 2023.

[7] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[8] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Efficient CTL verification via Horn constraints solving. In *HCVS@ETAPS*, volume 219 of *EPTCS*, pages 1–14, 2016.

[9] Sanjit Bhat and Hovav Shacham. Formal verification of the Linux kernel eBPF verifier range analysis, 2022.

[10] Armin Biere. The AIGER and-inverter graph (AIG) format version 20071012, 2007.

[11] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 2021.

[12] Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.

[13] Martin Bromberger, Irina Dragoste, Rasha Faqeh, Christof Fetzer, Markus Krötzsch, and Christoph Weidenbach. A Datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In Boris Konev and Giles Reger, editors, *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings*, volume 12941 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2021.

[14] Martin Bromberger, Alberto Fiori, and Christoph Weidenbach. Deciding the Bernays-Schoenfinkel fragment over bounded difference constraints by simple clause learning over theories. In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, pages 511–533. Springer, 2021.

[15] Martin Bromberger, Chaahat Jain, and Christoph Weidenbach. SCL(FOL) can simulate non-redundant superposition clause learning. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 134–152.

Springer, 2023.

[16] Martin Bromberger, Lorenz Leutgeb, and Christoph Weidenbach. Symbolic model construction for saturated constrained Horn clauses. In *FroCoS*, volume 14279 of *Lecture Notes in Computer Science*, pages 137–155. Springer, 2023.

[17] Martin Bromberger, Simon Schwarz, and Christoph Weidenbach. SCL(FOL) revisited, 2023.

[18] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795, pages 93–107. Springer, 2013.

[19] The Firejail Contributors. Firejail, 2020. https://github.com/netblue30/firejail [Accessed: 02/2024].

[20] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verimap: A tool for verifying programs through transformations. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 568–574, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[21] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[22] Christian Deacon and the XDP-Firewall contributors. XDP-Firewall, 2020. https://github.com/gamemann/XDP-Firewall/ [Accessed: 02/2024].

[23] Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. Seventeen provers under the hammer. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[24] The Linux Developers. The eBPF verifier, 2014. https://docs.kernel.org/bpf/verifier.html [Accessed: 02/2024].

[25] Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, and Andreas Podelski. Ultimate treeautomizer (CHC-COMP tool description). In *HCVS/PERR@ETAPS*, volume 296 of *EPTCS*, pages 42–47, 2019.

[26] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[27] John P. Gallagher and Bishoksan Kafle. Analysis and transformation tools for constrained Horn clause verification. *CoRR*, abs/1405.3883, 2014.

[28] 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 Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1069–1084. ACM, 2019.

[29] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.

[30] Alberto Griggio and Martin Jonás. Kratos2: An SMT-based model checker for imperative programs. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*, volume

13966 of *Lecture Notes in Computer Science*, pages 423–436. Springer, 2023.

[31] Arie Gurfinkel. Program verification with constrained Horn clauses (invited paper). In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 19–29, Cham, 2022. Springer International Publishing.

[32] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[33] Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. SMT-based verification of parameterized systems. In *SIGSOFT FSE*, pages 338–348. ACM, 2016.

[34] Jera Hensel and Jürgen Giesl. Proving termination of C programs with lists. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 266–285. Springer, 2023.

[35] Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn solver. In *FMCAD*, pages 1–7. IEEE, 2018.

[36] Hossein Hojjat, Philipp Rümmer, Pavle Subotic, and Wang Yi. Horn clauses for communicating timed systems. In *HCVS*, volume 169 of *EPTCS*, pages 39–52, 2014.

[37] Hsin-Wei Hung and Ardalan Amiri Sani. BRF: eBPF runtime fuzzer, 2023.

[38] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[39] Dejan Jovanovic and Clark W. Barrett. Polite theories revisited. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2010.

[40] Alan Jowett and the bpf-conformance contributors. eBPF conformance test suite, 2022. https://github.com/Alan-Jowett/bpf_conformance [Accessed: 02/2024].

[41] Toghrul Karimov, Engel Lefaucheux, Joël Ouaknine, David Purser, Anton Varonka, Markus A. Whiteland, and James Worrell. What's decidable about linear loops? *Proc. ACM Program. Lang.*, 6(POPL):1–25, 2022.

[42] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[43] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[44] K. Rustan M. Leino. This is Boogie 2, June 2008.

[45] K. Rustan M. Leino. Developing verified programs with Dafny. In Ben Brosgol, Jeff Boleng, and S. Tucker Taft, editors, *Proceedings of the 2012 ACM Conference on High Integrity Language Technology, HILT '12, December 2-6, 2012, Boston, Massachusetts, USA*, pages 9–10. ACM, 2012.

[46] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.

[47] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the security of Linux eBPF subsystem. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2023, Seoul, Republic of Korea, August 24-25, 2023*, pages 87–92. ACM, 2023.

[48] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories:

From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53:937–977, November 2006.

[49] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[50] Manfred Paul. CVE-2020-8835: Linux kernel privilege escalation via improper eBPF program verification, 2020. https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification [Accessed: 02/2024].

[51] Corneliu Popeea, Andrey Rybalchenko, and Andreas Wilhelm. Reduction for compositional verification of multi-threaded programs. In *FMCAD*, pages 187–194. IEEE, 2014.

[52] Siddharth Priya, Yusen Su, Yuyan Bao, Xiang Zhou, Yakir Vizel, and Arie Gurfinkel. Bounded model checking for LLVM. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 214–224. IEEE, 2022.

[53] Zvonimir Rakamaric and Michael Emmi. Smack: Decoupling source language details from verifier implementations. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014.

[54] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pages 495–507. Springer, 2019.

[55] Alexei Starovoitov, Daniel Borkmann, and the eBPF community. eBPF Documentation. https://ebpf.io/, 2014.

[56] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. Soltype: refinement types for arithmetic overflow in Solidity. *Proc. ACM Program. Lang.*, 6(POPL):1–29, 2022.

[57] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. Consort: Context- and flow-sensitive ownership refinement types for imperative programs. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 684–714. Springer, 2020.

[58] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: eBPF range analysis verification. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 226–251, Cham, 2023. Springer Nature Switzerland.

[59] Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In Harald Ganzinger, editor, *16th International Conference on Automated Deduction, CADE-16*, volume 1632 of *LNAI*, pages 314–328. Springer, 1999.

[60] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *22nd International Conference on Automated Deduction (CADE-22)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145, Montreal, Canada, August 2009. Springer.

[61] L. Wos. J. A. Robinson. Automatic deduction with hyper-resolution. International journal of computer mathematics, vol. 1 no. 3 (1965), pp. 227–234. *Journal of Symbolic Logic*, 39(1):189–190, 1974.

[62] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.

[63] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg, and Emmanuel Baccelli. End-to-end mechanized proof of an ebpf virtual machine for micro-controllers. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 293–316. Springer, 2022.

A Operational Semantics for Arithmetic-Logic Instructions

All arithmetic-logic instructions are encoded in the following fashion:

$$\varphi_{\text{op}} \mid\mid L_{pc}(r_0, r_1, \ldots, r_{10}, M) \implies L_{pc+1}(r_0, r_1', \ldots, r_{10}, M)$$

In Table 7, we give a comprehensive reference for the formula φ_{op} for all 64-bit arithmetic-logic instructions.

Remarks

- Without loss of generality, all registers are assumed to have r_1 as destination register. If applicable, r_0 is assumed as source register.
- The overview is given in SMT-LIB syntax. The variable `r1p` denotes variable r_1'.
- For brevity, 64-bit constants in SMT-LIB are abbreviated. A constant that is denoted by `#x2a` would expand to the 64-bit constant `#x000000000000002a` of the same value. Constants of other bit-lengths are written in full length.

B Operational Semantics for Control-Flow Instructions

All control-flow instructions are encoded in the following fashion:

$$\begin{aligned} \varphi_{\text{op}} \mid\mid L_{pc}(r_0, r_1, \ldots, r_{10}, M) &\implies L_{pc+\text{off}}(r_0, r_1', \ldots, r_{10}, M) \\ \neg\varphi_{\text{op}} \mid\mid L_{pc}(r_0, r_1, \ldots, r_{10}, M) &\implies L_{pc+1}(r_0, r_1', \ldots, r_{10}, M) \end{aligned}$$

Table 8 contains a comprehensive reference for the formula φ_{op} for all 64-bit control-flow instructions. The remarks from Appendix A apply.

C Operational Semantics for Memory Instructions

Memory Load Instructions Memory loading instructions are encoded as follows:

$$\varphi_{\text{op}} \mid\mid L_{pc}(r_0, r_1, \ldots, r_{10}, M) \implies L_{pc+1}(r_0, r_1', \ldots, r_{10}, M)$$

Table 9 contains the formula φ_{op} for all memory loading instructions.

Memory Store Instructions In contrast, memory storing instructions are encoded as

$$\varphi_{\text{op}} \mid\mid L_{pc}(r_0, r_1, \ldots, r_{10}, M) \implies L_{pc+1}(r_0, r_1, \ldots, r_{10}, M')$$

Table 10 gives a comprehensive reference for φ_{op} for all memory loading instructions. The variable `Mp` in SMT-LIB denotes variable M'. For both tables, the remarks from Appendix A apply.

Op.	Mnemonic	φ_{op}
0x7	r1 += imm	(= r1p (bvadd r1 imm))
0xf	r1 += r0	(= r1p (bvadd r1 r0))
0x17	r1 -= imm	(= r1p (bvsub r1 imm))
0x1f	r1 -= r0	(= r1p (bvsub r1 r0))
0x27	r1 *= imm	(= r1p (bvmul r1 imm))
0x2f	r1 *= r0	(= r1p (bvmul r1 r0))
0x37	r1 /= imm	(= r1p (ite (= imm #x00) #x00 (bvudiv r1 imm)))
0x3f	r1 /= r0	(= r1p (ite (= r0 #x00) #x00 (bvudiv r1 r0)))
0x47	r1 |= imm	(= r1p (bvor r1 imm))
0x4f	r1 |= r0	(= r1p (bvor r1 r0))
0x57	r1 &= imm	(= r1p (bvand r1 imm))
0x5f	r1 &= r0	(= r1p (bvand r1 r0))
0x67	r1 <<= imm	(= r1p (bvshl r1 (bvurem imm #x40)))
0x6f	r1 <<= r0	(= r1p (bvshl r1 (bvurem r0 #x40)))
0x77	r1 l>>= imm	(= r1p (bvlshr r1 (bvurem imm #x40)))
0x7f	r1 l>>= r0	(= r1p (bvlshr r1 (bvurem r0 #x40)))
0x87	r1 = -r1	(= r1p (bvneg r1))
0x97	r1 %= imm	(= r1p (bvurem r1 imm))
0x9f	r1 %= r0	(= r1p (bvurem r1 r0))
0xa7	r1 ^= imm	(= r1p (bvxor r1 imm))
0xaf	r1 ^= r0	(= r1p (bvxor r1 r0))
0xc7	r1 a>>= imm	(= r1p (bvashr r1 (bvurem imm #x40)))
0xcf	r1 a>>= r0	(= r1p (bvashr r1 (bvurem r0 #x40)))
0xb7	r1 = imm	(= imm r1p)
0xbf	r1 = r0	(= r1p r0)
0x18	r1 = imm	(= imm r1p)
0xd4	le16 r1	(= r1p r1)
0xd4	le32 r1	(= r1p r1)
0xd4	le64 r1	(= r1p r1)
0xdc	be16 r1	(= r1p (concat (concat #x000000000000 ((_ extract 7 0) r1)) ((_ extract 15 8) r1)))
0xdc	be32 r1	(let ((a!1 (concat (concat (concat #x00000000 ((_ extract 7 0) r1)) ((_ extract 15 8) r1)) ((_ extract 23 16) r1)))) (= r1p (concat a!1 ((_ extract 31 24) r1))))
0xdc	be64 r1	(let ((a!1 (concat (concat (concat ((_ extract 7 0) r1) ((_ extract 15 8) r1)) ((_ extract 23 16) r1)) ((_ extract 31 24) r1))))(let ((a!2 (concat (concat (concat a!1 ((_ extract 39 32) r1)) ((_ extract 47 40) r1)) ((_ extract 55 48) r1)))) (= r1p (concat a!2 ((_ extract 63 56) r1)))))

Table 7: The background formula φ_{op} for arithmetic-logic instructions.

Opcode	Mnemonic	φ_{op}
0x5	ja r1, 42	true
0x15	jeq r1, 42	(= r1 imm)
0x1d	jeq r1, r0	(= r1 r0)
0x25	jgt r1, 42	(bvugt r1 imm)
0x2d	jgt r1, r0	(bvugt r1 r0)
0x35	jgt r1, 42	(bvuge r1 imm)
0x3d	jgt r1, r0	(bvuge r1 r0)
0xa5	jlt r1, 42	(bvult r1 imm)
0xad	jlt r1, r0	(bvult r1 r0)
0xb5	jle r1, 42	(bvule r1 imm)
0xbd	jle r1, r0	(bvule r1 r0)
0x45	jset r1, 42	(distinct (bvand r1 imm) #x00)
0x4d	jset r1, r0	(distinct (bvand r1 r0) #x00)
0x55	jne r1, 42	(distinct r1 imm)
0x5d	jne r1, r0	(distinct r1 r0)
0x65	jsgt r1, 42	(bvsgt r1 imm)
0x6d	jsgt r1, r0	(bvsgt r1 r0)
0x75	jsge r1, 42	(bvsge r1 imm)
0x7d	jsge r1, r0	(bvsge r1 r0)
0xc5	jslt r1, 42	(bvslt r1 imm)
0xcd	jslt r1, r0	(bvslt r1 r0)
0xd5	jsle r1, 42	(bvsle r1 imm)
0xdd	jsle r1, r0	(bvsle r1 r0)

Table 8: The background formula φ_{op} for control-flow instructions.

Op.	Mnemonic	φ_{op}
0x71	r1 = *(u8 *) r0	(= r1p (concat #x00000000000000 (select M r0)))
0x69	r1 = *(u16 *) r0	(= r1p (concat #x000000000000 (select M (bvadd #x01 r0)) (select M r0)))
0x61	r1 = *(u32 *) r0	(= r1p (concat #x00000000 (select M (bvadd #x03 r0)) (select M (bvadd #x02 r0)) (select M (bvadd #x01 r0)) (select M r0)))
0x79	r1 = *(u64 *) r0	(= r1p (concat (select M (bvadd #x07 r0)) (select M (bvadd #x06 r0)) (select M (bvadd #x05 r0)) (select M (bvadd #x04 r0)) (select M (bvadd #x03 r0)) (select M (bvadd #x02 r0)) (select M (bvadd #x01 r0)) (select M r0)))

Table 9: The background formula φ_{op} for memory load instructions.

Op.	Mnemonic	φ_{op}
0x72	`*(u8 *) r1 = imm`	`(= Mp (store M r1 #x2a))`
0x6a	`*(u16 *) r1 = imm`	`(= Mp (store (store M r1 #x2a) (bvadd #x01 r1) #x00))`
0x62	`*(u32 *) r1 = imm`	`(let ((a!1 (store (store (store (store M r1 #x2a) (bvadd #x01 r1) #x00) (bvadd #x02 r1) #x00) (bvadd #x03 r1) #x00))) (= Mp a!1))`
0x7a	`*(u64 *) r1 = imm`	`(let ((a!1 (store (store (store (store M r1 #x2a) (bvadd #x01 r1) #x00) (bvadd #x02 r1) #x00) (bvadd #x03 r1) #x00)))(let ((a!2 (store (store (store a!1 (bvadd #x04 r1) #x00) (bvadd #x05 r1) #x00) (bvadd #x06 r1) #x00))) (= Mp (store a!2 (bvadd #x07 r1) #x00))))`
0x73	`*(u8 *) r1 = r0`	`(= Mp (store M r1 ((_ extract 7 0) r0)))`
0x6b	`*(u16 *) r1 = r0`	`(= Mp (store (store M r1 ((_ extract 7 0) r0)) (bvadd #x01 r1) ((_ extract 15 8) r0)))`
0x63	`*(u32 *) r1 = r0`	`(let ((a!1 (store (store (store M r1 ((_ extract 7 0) r0)) (bvadd #x01 r1) ((_ extract 15 8) r0)) (bvadd #x02 r1) ((_ extract 23 16) r0)))) (= Mp (store a!1 (bvadd #x03 r1) ((_ extract 31 24) r0))))`
0x7b	`*(u64 *) r1 = r0`	`(let ((a!1 (store (store (store M r1 ((_ extract 7 0) r0)) (bvadd #x01 r1) ((_ extract 15 8) r0)) (bvadd #x02 r1) ((_ extract 23 16) r0))))(let ((a!2 (store (store (store a!1 (bvadd #x03 r1) ((_ extract 31 24) r0)) (bvadd #x04 r1) ((_ extract 39 32) r0)) (bvadd #x05 r1) ((_ extract 47 40) r0)))) (= Mp (store (store a!2 (bvadd #x06 r1) ((_ extract 55 48) r0)) (bvadd #x07 r1) ((_ extract 63 56) r0)))))`

Table 10: The background formula φ_{op} for memory store instructions.