Master's Thesis

Verification Condition Generation for eBPF

Troels Korreman Nielsen (xck773)
troels.korreman@gmail.com

Supervisor: Ken Friis Larsen

PCC and eBPF
Department of Computer Science
University of Copenhagen

November 22, 2024

Abstract

The eBPF subsystem for Linux allows loading programs at runtime from user space into kernel space for execution. Using static analysis and JIT-compilation, both high performance and safety can be guaranteed. However, there is a conflict of interest between the desire to expand the capabilities of the eBPF program verifier and the need to manage its complexity. Proof-carrying code (PCC) is an alternative approach to program validation. It shifts the burden of validating safety to the code producer by requiring that a safety proof be submitted with every program. By replacing the static analyzer with a proof checker, consumers reduce their workload while producers can submit a wider range of programs. In this project, the prospect of using PCC to validate eBPF is explored with a focus on branching and looping programs. Subsets of the eBPF instruction set and safety policies are chosen as targets. Verification condition generators (VC-gen) for partial and total correctness are designed for these targets. A proof of concept VC-gen for partial correctness is implemented and compared to the eBPF verifier on synthetic programs. By dispatching verification conditions to a solver, PCC is shown to be able to validate several safe programs that the verifier rejects.

Contents

1 Introduction

The eBPF subsystem for Linux is a technology that allows extending the kernel dynamically while ensuring safety, security, and high performance[4]. While it is possible to load kernel modules at runtime, eBPF provides a safer and more secure method for instrumentation. This is accomplished through sand-boxing methods and static analysis techniques, which respectively restrict access to kernel space and validate that modules follow safety policies. With modules being written in a simple RISC-like instruction language, they can be JIT-compiled into efficient machine code on several different architectures.

eBPF is being used for a variety of applications such as packet filtering, activity monitoring, load balancing, kernel tracing, and much more. While originating as a Linux subsystem, the use of eBPF has been spreading to other operating systems and technologies. The technology is gaining traction in Microsoft with the "eBPF for Windows" project, as well as directly in hardware with projects such as hBPF.

Running user code in a privileged context such as kernel space introduces challenges to the safety and security of the kernel. The Linux subsystem therefore only runs eBPF modules that have been validated by the eBPF verifier, an embedded static analysis engine which rejects any programs that might compromise the kernel. This approach to validation has an unfortunate downside; the range of programs that may run is limited by the capabilities of the in-kernel verifier. Kernel developers are thus faced with the task of finding a compromise between maximizing the range of supported programs while minimizing the scope of the verifier to ensure its correctness.

In this project, we explore the possibility of validating eBPF programs using a proof-carrying code (PCC) architecture as an alternative to in-kernel static analysis. This approach would require users to submit a proof of adherence to a safety policy along with each module, replacing the in-kernel static analyzer with an in-kernel proof checker. By pushing the responsibility of ensuring safety to the user, the complexity of the in-kernel validation can be reduced, while the range of programs that may be validated can be greatly expanded. Focus is placed mainly on the process of *verification condition generation* (VC-gen), the construction of proof obligations that must be satisfied for a corresponding module to be considered safe. The aim is to demonstrate a combination of VC-gen and automated solving in userspace as a viable approach to eBPF validation.

1.1 Contributions

- A specification of the general safety requirements for running eBPF in kernel-space.
- A proof system for a subset of eBPF that includes branching instructions.
- A specification of a verification condition generator for that eBPF subset.
- A proof-of-concept implementation of the VC-generator.
- A collection of synthetic eBPF programs demonstrating the VC-generator.
- A proposed method for proving total correctness of cyclic eBPF programs.

2 eBPF subsystem for Linux

2.1 Architecture

The eBPF subsystem works by letting users load eBPF modules into the kernel and attach them to events in various kernel subsystems. Examples of such events include the arrival of network packets, tracing events, and classification events, with new event types being a future possibility[7].

Before a module can be attached to anything, the in-kernel verifier statically determines whether it terminates and is safe to execute. We will refer to this verifier as the kernel verifier. It only accepts programs that it deems to be completely safe with certainty, and is thus very conservative.

eBPF programs are written using an instruction set reminiscent of machine ISAs. With the eBPF instruction set being so close to a real machine language, it can be easily translated into machine code for different architectures. This allows users to write programs in a high-level language of choice and compile their programs into eBPF. That being said, modules are heavily limited in capabilities, both as a consequence of the verifier and the execution context being kernel space. The eBPF instruction set can be thought of as running on a virtual machine. It contains instructions for the following classes of functionality:

- Arithmetic on 32-bit and 64-bit integers.
- Load / store memory operations.
- Atomic operations on memory.
- Direct and conditional direct jumps.
- Calling system-defined helper functions.

Helper functions are used to define a range of data structures and extensions that any given module may interact with. One of the primary uses of helpers is to provide interaction with *eBPF maps*, safe data structures that are accessible to both users and modules, providing an avenue of interaction between the two spaces.

There are some typical features that are missing from eBPF programs, as a result of both verifier limitations and the properties of kernel space. Most importantly, there are no memory allocations and no instructions to perform indirect jumps. The available memory is defined by a context object provided to the module during execution and a 512-byte frame buffer.

The strict requirements for safety and performance combined with a limited set of features makes eBPF an ideal use target for both static analysis methods and proof systems.

2.2 Safety requirements

Running user-provided programs in kernel space is necessarily going to result in a whole slew of challenges relating to safety and security. It could be argued that a primary advantage of eBPF is how these challenges are solved in a step before any eBPF code is ever run. The challenges stem mainly from two sources.

First there are the typical challenges to safety that arise from running code in kernel space; eBPF modules aren't executed as processes and thus aren't given any of the safety nets provided to a process. There is no scheduling, no interrupt handling, no panicking, no memory virtualization, and so on. This means that in order for programs to execute safely, they must run correctly in a number of ways that the process abstraction usually take care of. Performing unsafe operations incorrectly can result in corruption of the kernel state, compromising it in any number of unknown ways.

The second source of challenge is security. Raw code that runs in kernel space has unchecked access to kernel memory and can directly communicate with userspace processes. Since eBPF can be run from non-root processes in some cases, it is crucial that modules cannot be used to intentionally exploit the

kernel or leak secrets. In the same vein, it shouldn't be possible to indirectly exploit the kernel through buggy eBPF modules.

In this case, many security concerns can be considered safety concerns and vice versa. The verifier enforces that modules behave correctly, respecting both of these concerns. Through documentation[7] and source code we can build a model of the policies that it enforces. However, the verifier is stricter than the policies it is designed to enforce. It is therefore a good idea to develop a model of the actual safety policies we wish to enforce.

Since the "intended" safety policies aren't well-documented, the best we can do is to make an educated guess based on the verifier itself. We decide on the following rules, defining a policy that ensures both kernel safety and security. We'll refer to these as our safety policy.

- Modules may not contain unreachable code.
- Execution must terminate within a reasonable time-frame.
- Execution must never reach a state from which it cannot continue.
- Memory may not be touched unless within a region given access to.
- Memory access must be aligned.
- Memory addresses must not be leaked.
- Memory data must not be leaked unless permitted.
- Registers must be initialized before they are read.
- Helper functions must be called with valid parameters (file descriptors, etc.)

2.3 Verifier

The task of enforcing the previously stated rules is non-trivial, in fact, it is impossible to decide for several of the problems (due to the halting problem). Other requirements are simply so complex that perfectly checking all programs becomes infeasibly slow. Since we cannot allow any unsafe modules to run, the best the verifier can do is to accept some subset of all valid modules. A significant remainder of safe modules must be rejected along with all of the unsafe ones, the result being that developers of the eBPF subsystem must perform a balancing act between keeping the verifier within a manageable level of complexity and supporting a satisfactory set of modules.

The verifier validates modules through a combination of graph analysis, path enumeration, typing, and abstract interpretation[1]. First, the module is analyzed as a control graph. Originally the graph was validated to be a DAG, but since Linux version 5.3 some support for bounded loops has been implemented[12]. Instead, the graph is now verified to only contain cycles in some restricted way. At this point, the graph is also checked for obviously unreachable code. After this, the module is analyzed using a kind of abstract interpretation through path enumeration. The analysis essentially works by tracking several states through the program and splitting states when branching instructions are encountered. Each state object tracks a variety of abstract information including types, upper and lower bounds of registers, and known bits of registers. Rather than employing a lattice with merging and widening, previous states are remembered for each instruction and states that represent subsets of a previous ones are pruned. When encountering an operation that may be unsafe, the correctness of parameters must be inferred from the abstract branch state. In order to keep the running time and memory usage of verification within reasonable bounds, limits are imposed on the amount of branching states and the total amount of instructions that may be analyzed. Since v5.2, the total instruction limit has been set to one million[12].

Putting this together, the kernel verifier is rather complex and has some significant limitations. The tracking of register values is simple. The path enumeration approach can be susceptible to path explosion, especially with the added support for verifying bounded loops. Another point of contention

is that the verifier lacks a formal foundation. There is no specification that allows users to know beforehand whether a given program will validate, nor is there any proof of soundness. The best source of knowledge about the verifier is the source code of the verifier itself, as documentation is somewhat sparse and outdated.

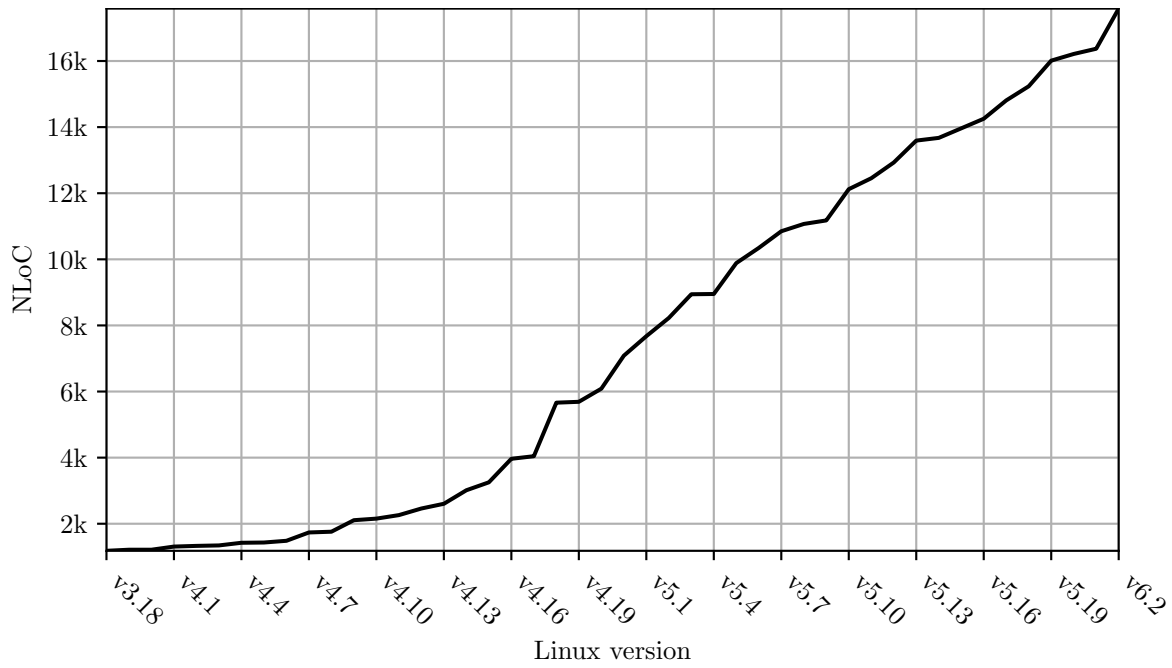
2.4 Growing complexity

The results of the push for an evermore feature-rich verifier are clearly visible in kernel code. A 2019 blog post[3] does an analysis of the verifier from a complexity standpoint. Using the Lizard code complexity analyzer[14], the verifier is analyzed through a few metrics:

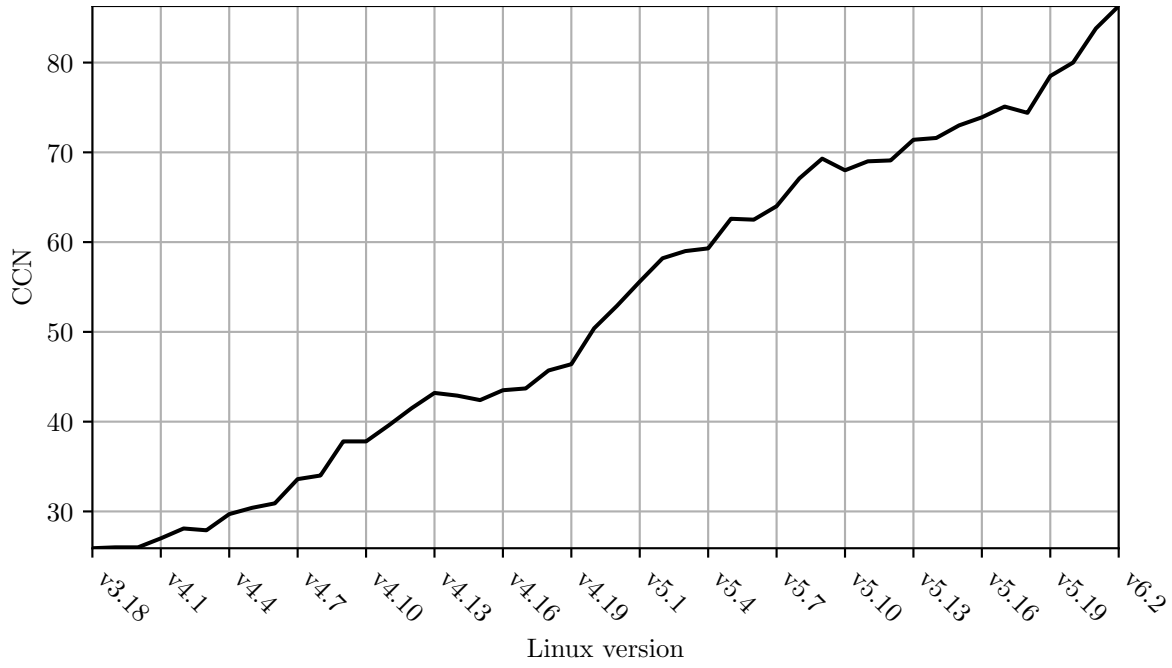
- Code size measured by the number of lines of code (NLoC), excluding comments and white-space.
- Code size as a percentage of the entire eBPF subsystem.
- Average cyclomatic complexity[9] of the worst-scoring functions.

While there is no measure that can objectively quantify the complexity of a code base, these measures still give us an indication as to the current practical state of affairs. A clear increase in the NLoC and cyclomatic complexity is observed, while the relative code size remains relatively stable between 20-40%. However, this last measure indicates that the verifier is consistently expanding along with the rest of the system as features are added.

Repeating the experiments using the same analysis software, we can observe how the verifier has continued to develop, shown in fig. 1. Neither the increase in code size nor cyclomatic complexity has slowed down since the original observations were made.



(a) Total number of lines of code, excluding empty lines and comments.



(b) Cyclomatic complexity score as an average of the ten most complex functions.

Figure 1: Rough complexity measures of the eBPF verifier as a function of Linux versions. Both `kernel/bpf/verifier.c` and its dependency `kernel/bpf/btf.c` are included in the measure.

3 Proof-carrying code

The kernel verifier is just one solution to the problem of running an inherently untrusted eBPF module inside the privileged kernel space. So we have a situation where code is untrusted both in the sense that an attacker might use eBPF to exploit the kernel, and that a developer may not trust their own code to be safe. In either case, our goal is to ensure that a piece of untrusted code can nevertheless be run inside of a sensitive context without compromising safety. Furthermore, we wish to uphold safety without sacrificing performance, as performance is one of the primary reasons that we wish to run code in the kernel space in the first place.

Another approach to solving the problem of trusting untrusted programs is proof-carrying code (PCC) [11]. In a PCC architecture, the code producer must supply a proof of safety along with every program they wish to run. The code consumer then only has to verify correctness of the safety proof with respect to the program, after which the program may be run any number of times. If the proof is valid, the program can be run safely without further validation. If the proof isn't valid, the program will simply be rejected.

The overall architecture of a PCC solution for eBPF is shown in fig. 2 and compared to a generalization of the current architecture. A safety policy (including a proof system) must be designed and published by the consumer, in this case kernel developers. This policy is exposed to the producer and acts as an interface. It defines the safety requirements for eBPF modules as well as a system for proving safety. Using a variety of methods and frameworks, the producer creates an eBPF module as well as a proof of correctness. Both are submitted to eBPF subsystem. Here, the proof is checked to ensure that the module adheres to the safety policy. Once safety has been established, the module can be executed through interpretation or JIT-compilation.

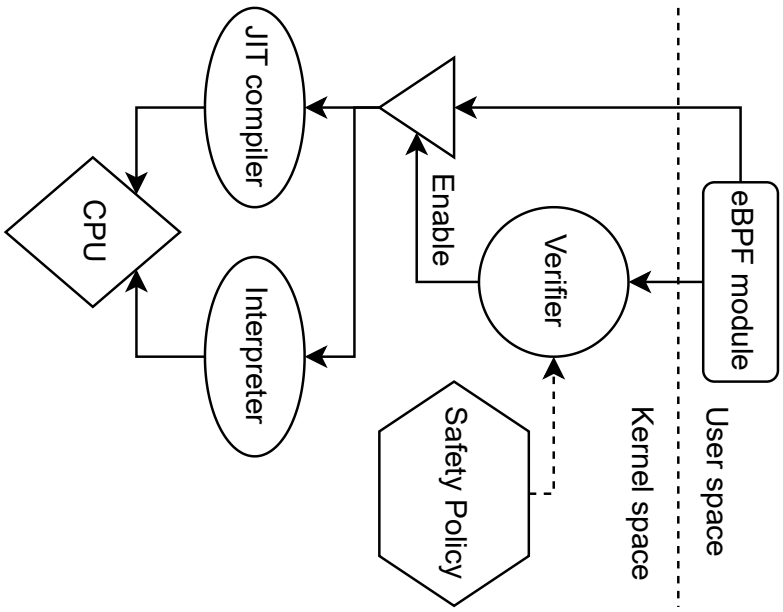
The PCC approach has a number of advantages. The primary advantage is that it significantly lightens the workload for the consumer. There is no need to implement and maintain an analysis engine. Instead, it is only necessary to define a proof system, ensure its soundness, and implement a proof checker. Forcing consumers to clearly define the criteria for safety is beneficial to both sides of the trust boundary. Meanwhile, maintaining a proof checker is generally going to be a simpler task in terms of complexity, runtime performance, and code size. Finally, as with static validation, safety is ensured before code is run, so there is no need to perform runtime checks that can impede performance.

Of course, the more difficult task of proving correctness for programs doesn't simply disappear along with the verifier. Rather, that responsibility is moved from the consumer to the producer. There are a number of ways to produce safety proofs, ranging from fully manual proof writing to fully-automated proof generation. While it may seem like this makes module development more cumbersome, the task is in principle not much more difficult than when it was being automated inside the kernel. The in-kernel verifier is in essence an automated solver, it shouldn't be impossible to move that solver to userspace. The major difference is that while the trusted verifier could simply report whether it found a module to be safe, a similar userspace verifier must support its findings with a proof.

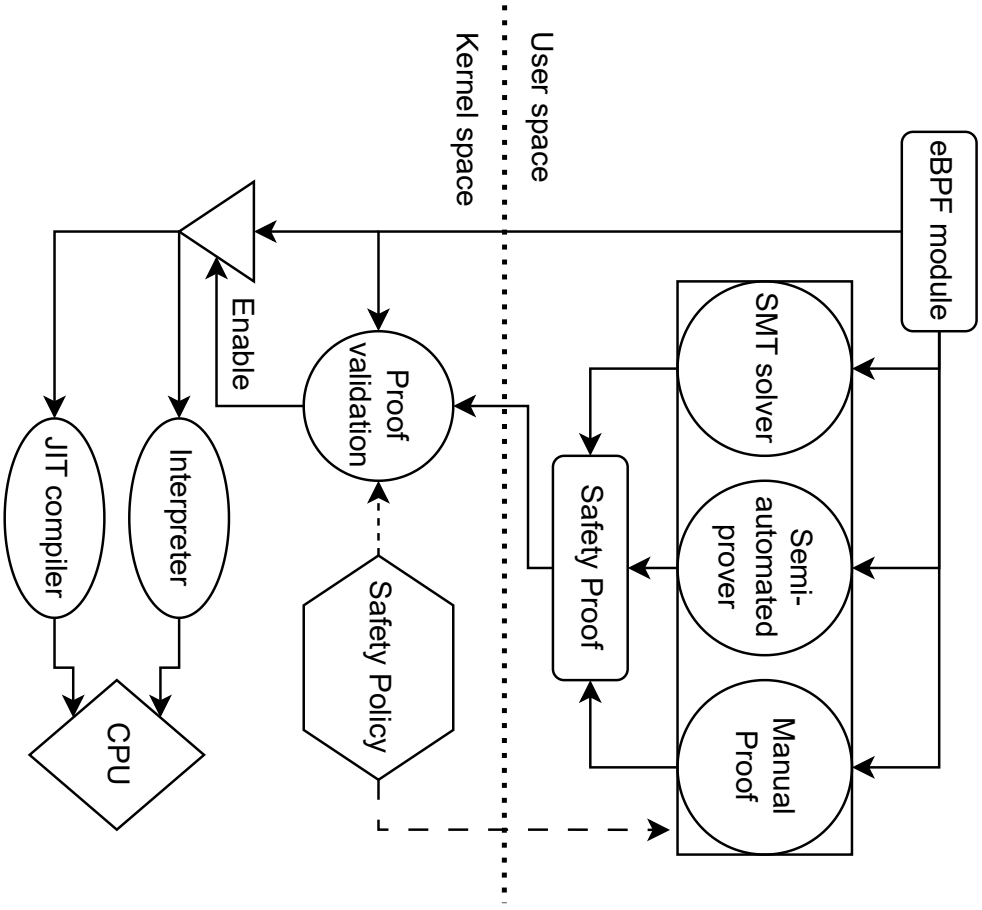
Our method for checking compliance with the safety policy is based on verification condition generation (VC-gen). Based on a program, we generate a set of verification conditions to serve as proof obligations. If they can be proved, there must exist a safety proof for the program, and the program must therefore be safe. The VC-gen should be openly specified, as both the producer and consumer must be able to compute the conditions. Otherwise, the producer wouldn't be able to perform offline certification. Our primary focus is on the challenges for VC-gen associated with conditional jumps to static locations.

3.1 Hoare logic

In order to create PCC systems, we need a way to logically reason about programs. To that end we use Hoare triples[8], which allow us express logical properties of program elements by specifying pre- and post-conditions for statements. A triple $\{P\}\ s\ \{Q\}$ expresses that, if a pre-condition P is true for a program state Σ_a, the post-condition Q will be true for the state Σ_b that results from executing the statement s on Σ_a. To create a full proof system for a programming language, we must construct



(a) Current verifier architecture.



(b) Alternative PCC architecture.

Figure 2: Datapath overviews of the verifier architecture and proposed PCC alternative.

a *Hoare logic*, a set of inference rules that allow us to derive Hoare triples. Our logic can be said to be sound if all derivable triples are valid. While showing soundness of the logic is a goal, achieving completeness for program logics is a more difficult goal that we ignore.

In the context of PCC, safety policies can be encoded in the Hoare logic. Our criteria for soundness is that any program which can be derived using the logic is safe. We aren't necessarily concerned with proving correctness of algorithms, as our goal is only to show that programs follow safety policies while executing. However, the safety of programs could rely on correctness of its subparts, so it comes out to the same in the end.

3.2 Weakest pre-condition calculus

In order to automate the process of proving validity, we use a weakest precondition calculus inspired from [8]. The basic idea is to specify a function that generates the weakest possible precondition such that the postcondition is still satisfied. Formally, we wish to define a function (WP : $\mathit{Prog} \times \mathit{Formula} \rightarrow \mathit{Formula}$) such that the triple $\{\mathrm{WP}(s,\ Q)\}\ s\ \{Q\}$ is valid.

This method can be used as a mechanism for generating verification conditions. Given a program s, let $P = \mathrm{WP}(s, \top)$. Knowing that the WP-calculus is sound, the triple $\{P\}\ s\ \{\top\}$ must be valid. In other words, if P is satisfied, then the program is safe to run, and proving P is equivalent to showing that the program is valid for all initial program states. If we ensure that formulas such as P are expressed in a well-supported logic, we can pass them as proof obligations to automated solvers.

4 Formalizing eBPF proofs

In order to design a VC-generator, a logic for deriving correct eBPF programs must first be defined. This section details the grammars for programs, expressions, formulas, the operational semantics, and the inference rules for a Hoare logic that derives valid programs.

4.1 eBPF grammar

The grammar we work with is shown in fig. 3. We choose a subset of the eBPF instruction set[1] to work with, in order to simplify matters when developing inference rules and a VC-generator. It should still be sufficient for demonstrating several aspects of VC-gen and PCC as a whole. The language is limited to a subset of 64-bit arithmetic operations, load and store instructions, and branching control flow instructions. This excludes several features including atomic operations, including 32-bit operators, calling helper functions, and specialized instructions such as **LoadMapFd**.

Since eBPF doesn't support indirect jumps, continuations can only pass on control to statically defined locations. This means that a module can be viewed as a *control graph*, nodes represented by instructions and edges denoting control passing from each instruction to the next.

A large set of instructions are *straightline* instructions, meaning that they will always pass on control to the next instruction in the sequence. Using this to our advantage, we represent eBPF modules as collections of control-passing code blocks. Each block consists of a unique label and a sequence of straightline instructions (*statements*) ending in a control-passing instruction (a *continuation*). A module is thus represented as a control graph of blocks rather than a single sequence of instructions.

The control graph representation allows us to separate concerns in two for our semantics, logic, and VC-gen. There is the concern of modeling sequences of statements, and then there is the separate concern of resolving interconnected control graphs.

4.2 Values, expressions, and formulas

While eBPF instructions cannot represent expressions, they are still necessary for writing proofs. One thing to be aware of is that eBPF instructions (and expressions by extension) operate on 64-bit bit strings, not on unbounded integers. Regular arithmetic operations such as addition, subtraction, and division are subject to overflows and underflows. Real world eBPF also supports bitwise operations and will readily treat a value as signed in one instance and unsigned in the next. Memory operations use various widths, this is simply handled by ignoring and zeroing upper bits where relevant. To make it clear that these operations are functions on bit-strings, they are written as regular functions. For example, the triangular sum of n would be rewritten as:

$$n(n-1)/2 = \mathbf{div}(\mathbf{mul}(n, \mathbf{sub}(n, 1)), 2)$$

While the kernel verifier may treat overflows and underflows as errors, these functions are actually total, with the exception of division and modulo operations. Overflows are often used intentionally as well, so making them illegal would be unnecessarily restrictive. Instead, we treat arithmetic operations as functions $\mathbb{Z}^{64\circlearrowleft} \times \mathbb{Z}^{64\circlearrowleft} \to \mathbb{Z}^{64\circlearrowleft}$, where $\mathbb{Z}^{64\circlearrowleft}$ is the set of 64-length bit-strings.

We must also choose a logic language to be used for expressing pre- and postconditions in. Here we go with a general first-order language with quantifiers and comparisons, and assume the usual inference rules associated with this. Similarly to arithmetic, comparisons are performed on values in $\mathbb{Z}^{64\circlearrowleft}$, so we model them as functions of the type $\mathbb{Z}^{64\circlearrowleft} \times \mathbb{Z}^{64\circlearrowleft} \to \{\mathbf{true}, \mathbf{false}\}$. The logic also contains one more unusual construct which is used to express that a pair of values represents the address and size of an accessible buffer. There are no extra rules for deriving **is_buffer**, it has to be introduced in the requirements of a module.

The full expression and formula grammars are shown in fig. 4.

$$
\begin{array}{rcl}
\mathit{module} & ::= & \textbf{module} : \textbf{start}\ \mathit{label}\ \textbf{blocks}\ (\mathit{block}\ ,)^*\ \textbf{end} \\
\mathit{block} & ::= & \textbf{block}\ \mathit{label}\ (\mathit{stmt}\ ;\ )^*\ \mathit{cont} \\
\mathit{stmt} & ::= & \textbf{unop}\ \mathit{unalu}\ \mathit{reg} \\
 & | & \textbf{binop}\ \mathit{binalu}\ \mathit{reg}\ \mathit{regimm} \\
 & | & \textbf{store}\ \mathit{memsize}\ \mathit{memref}\ \mathit{regimm} \\
 & | & \textbf{load}\ \mathit{memsize}\ \mathit{reg}\ \mathit{memref} \\
 & | & \textbf{assert}\ \mathit{formula} \\
\mathit{cont} & ::= & \textbf{exit} \\
 & | & \textbf{jmp}\ \mathit{label} \\
 & | & \textbf{jcc}\ \mathit{cc}\ \mathit{reg}\ \mathit{regimm}\ \mathit{label}\ \mathit{label} \\
\mathit{regimm} & ::= & \mathit{reg} \mid \mathit{imm} \\
\mathit{memref} & ::= & \texttt{[}\ \mathit{reg}\ ((+ \mid -)\ \mathit{imm})?\ \texttt{]} \\
\mathit{memsize} & ::= & 1 \mid 2 \mid 4 \mid 8 \\
\mathit{reg} & ::= & \textbf{r0} \mid \textbf{r1} \mid \textbf{r2} \mid \textbf{r3} \mid \textbf{r4} \mid \textbf{r5} \mid \textbf{r6} \mid \textbf{r7} \mid \textbf{r8} \mid \textbf{r8} \mid \textbf{r9} \mid \textbf{r10} \\
\mathit{cc} & ::= & \textbf{eq} \mid \textbf{ne} \mid \textbf{gt} \mid \textbf{ge} \mid \textbf{lt} \mid \textbf{le} \\
\mathit{binalu} & ::= & \textbf{mov} \mid \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div} \mid \textbf{mod} \\
\mathit{unalu} & ::= & \textbf{neg} \\
\mathit{label} & ::= & \langle\ \mathit{identifier}\ \rangle \\
\mathit{imm} & ::= & \langle\ \mathit{64\text{-}bit\ literal}\ \rangle
\end{array}
$$

Figure 3: Grammar for our chosen subset of eBPF.

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{num} \\
 & | & \mathit{var} \\
 & | & \mathit{unalu}(\mathit{expr}) \\
 & | & \mathit{binalu}(\mathit{expr}, \mathit{expr}) \\
\mathit{var} & ::= & \langle\ \mathit{identifer}\ \rangle \\
\mathit{num} & ::= & \langle\ \mathit{64\text{-}bit\ literal}\ \rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathit{form} & ::= & \textbf{true} \\
 & | & \textbf{false} \\
 & | & \mathit{form} \wedge \mathit{form} \\
 & | & \mathit{form} \vee \mathit{form} \\
 & | & \mathit{form} \implies \mathit{form} \\
 & | & \exists \mathit{var}.\ \mathit{form} \\
 & | & \forall \mathit{var}.\ \mathit{form} \\
 & | & \mathit{cc}(\mathit{expr}, \mathit{expr}) \\
 & | & \textbf{is\_buffer}(\mathit{expr}, \mathit{expr})
\end{array}
$$

Figure 4: Expression and formula grammars.

4.3 Operational semantics

We won't be spending too much time on formalizing the operational semantics of eBPF, but at least an informal outline of the execution model is necessary. The eBPF grammar can be viewed as though it executes on a virtual RISC-like machine. Execution runs as follows:

1. The block matching the entry label of the module is pointed to by B.
2. The block in B is executed by resolving all statement in sequence.
3. Next, the continuation of the the block in B is resolved:
   - If the continuation is a jump, B is pointed on the jump target.
   - If the continuation is an exit, execution is terminated.
4. Go to 2.

Statements are resolved just as straightline instructions are in typical ISAs like x86, ARM, and RISC-V. Arithmetic operations replace the lefthand operand, and memory operations read from and write to an integer-addressed memory space. Assertions are resolved as is typical in high level languages:: if the assertion fails, execution is unable to continue.

4.4 Inference rules

When designing the inference rules for the core proof system, it's important to be aware of which safety policies we wish to enforce. Only two policies are chosen for this VC-generator:

- Arithmetic: Division and modulo operations must use non-zero divisors.
- Memory: Memory accesses must reference valid memory cells.

Some other policies would be fairly straightforward to include, such as memory alignment. Other policies like data leakage prevention would require more work.

The inference rules for partial correctness (shown in fig. 5) can be split into sets of rules for straightline instructions, composition, continuations, and modules.

Straightline rules The straightline rules are based on inference rules presented in [8]. Assignments occur through arithmetic operations, and registers are treated as regular variables being replaced in the precondition on assignment.

For instructions that perform memory operations, it must be shown that the address used is a valid address. Specifically, it must be shown to be a value within the bounds of an accessible buffer known through **is_buffer**. A quite severe limitation of this iteration of the logic is that memory is treated as a black box. Storing a value places no requirements on the pre- and postconditions, while loading a value overwrites the target register so that it can be any value. As a result, any module for which correctness depends on the contents of memory cannot be validated.

Finally, assertions must be proved in order to derive them.

Composition rules The composition rules allow two things: concatenation of statement sequences and relaxation of conditions. Together, these allow statements to be chained together with intermediate logic.

Continuations rules Since the dependencies between blocks are not acyclic, we run into an ordering problem when trying to create a set of rules for continuations. Taking inspiration from [10], we use an auxiliary mapping from labels to preconditions:

$$\Gamma : \mathit{label} \to \mathit{formula}$$

The derivation of a module must happen with respect to some such mapping Γ. We also define a module-wide postcondition Γ_Q, which doesn't have a corresponding block. This mapping of requirements allows us to retrieve predefined preconditions for blocks, which can in turn be used to derive postconditions for continuations. Postcondition derivations are expressed as sentences of the form $\Gamma \vdash j \curvearrowright Q$.

Module rules The module rules allow the construction of blocks from statements and continuations, and the construction of a module from a collection of blocks.

Blocks may be derived with respect to some Γ from derivations of their statement sequences and continuations. A collection of blocks may be derived if each block has a unique label in Γ, to ensure that there is no ambiguity about block transitions. From a collection of blocks a module can then be derived if its pre- and postconditions are compatible with its entry label and Γ_Q.

Straightline

UnAlu	$\{P[r \leftarrow A]\}\ \mathbf{unop}\ a\ r\ \{P\}$ where $A = \mathrm{alu}(a, r)$
BinAlu	$\{P[r \leftarrow A]\}\ \mathbf{binop}\ a\ r\ v\ \{P\}$ if $a \notin \{\mathbf{div}, \mathbf{mod}\}$ where $A = \mathrm{alu}(a, r, v)$
BinAluDiv	$\{v \neq 0 \wedge P[r \leftarrow A]\}\ \mathbf{binop}\ a\ r\ v\ \{P\}$ if $a \in \{\mathbf{div}, \mathbf{mod}\}$ where $A = \mathrm{alu}(a, r, v)$
Load	$\{\mathit{valid\_addr}(m) \wedge \forall v.\ P[r \leftarrow v]\}\ \mathbf{load}\ k\ r\ m\ \{P\}$
Store	$\{\mathit{valid\_addr}(m) \wedge P\}\ \mathbf{store}\ k\ m\ v\ \{P\}$
Assert	$\{A \wedge P\}\ \mathbf{assert}\ A\ \{P\}$

Composition

Seq	$\dfrac{\{P\}\ s_1\ \{Q\} \quad \{Q\}\ s_2\ \{S\}}{\{P\}\ s_1; s_2\ \{S\}}$
Cons	$\dfrac{\{P'\}\ s\ \{Q'\}}{\{P\}\ s\ \{Q\}}$ if $\forall \mathbf{r0} \ldots \mathbf{r10}.\ P \implies P' \wedge Q' \implies Q$

Continuations

Exit	$\Gamma \vdash \mathbf{exit} \curvearrowright \Gamma_Q$
Jmp	$\Gamma \vdash \mathbf{jmp}\ l \curvearrowright \Gamma(l)$
Jcc	$\Gamma \vdash \mathbf{jcc}\ c\ r\ v\ l_\mathtt{T}\ l_\mathtt{F} \curvearrowright Q \wedge \Gamma(l_\mathtt{T}) \vee \neg Q \wedge \Gamma(l_\mathtt{F})$ where $Q = \mathrm{cmp}(c, r, v)$

Modules

Block	$\dfrac{\Gamma(l) = P \quad \{P\}\ s\ \{Q\} \quad \Gamma \vdash j \curvearrowright Q}{\Gamma \vdash \mathbf{block}\ l\ s\ j}$ if $\forall \mathbf{r0} \ldots \mathbf{r10}.\ P$
ManyBlocks	$\Gamma \vdash \mathbf{blocks}\ \beta$ if all labels l in Γ correspond to a block in β
Module	$\dfrac{\Gamma \vdash \mathbf{blocks}\ \beta \quad \Gamma_Q = Q}{\Gamma \vdash \{P\}\ \mathbf{module} : \mathbf{start}\ l\ \mathbf{blocks}\ \beta\ \{Q\}}$ if $\forall \mathbf{r0} \ldots \mathbf{r10}.\ P \implies \Gamma(l)$

Where the validity of an address is denoted by:

$$\mathit{valid\_addr}(m) = \exists p, s.\ \mathbf{is\_mem}(p, s) \wedge p \leq m < p + s - (k - 1)$$

Figure 5: Inference rules for partial correctness of eBPF modules.

5 Verification condition generation

Having formalized a proof system, we can now move on to define our verification condition generation algorithm. We are concerned mainly with soundness of the algorithm rather than completeness; some perfectly safe modules may not be provable, but no unsafe modules should be provable.

5.1 WP-calculus

For code block bodies that consist purely of straight-line instructions, we can compute VCs using a weakest precondition calculus. Here we don't need to concern ourselves with the retrieval of post-conditions, instead we can focus purely on their transformation to pre-conditions.

$$
\begin{array}{lcl}
\mathrm{WP}(s\ ;\ b,\ \ Q) & = & \mathrm{WP}(s, \mathrm{WP}(b,\ Q)) \\
\mathrm{WP}(\textbf{assert}\ A,\ Q) & = & A\ \texttt{\&\&}\ Q \\
\mathrm{WP}(\textbf{unop}\ alu\ r,\ Q) & = & \forall v.\ v = alu(r) \implies Q[r \leftarrow v] \\
\mathrm{WP}(\textbf{binop}\ alu\ r\ v,\ Q) & = & \forall v.\ v = alu(r, v) \implies Q[r \leftarrow v] \\
 & & \text{if } alu \notin \{\textbf{mod}, \textbf{div}\} \\
\mathrm{WP}(\textbf{binop}\ alu\ r\ v,\ Q) & = & v \neq 0 \wedge \forall v.\ v = alu(r, v) \implies Q[r \leftarrow v] \\
 & & \text{if } alu \in \{\textbf{mod}, \textbf{div}\} \\
\mathrm{WP}(\textbf{store}\ k\ m\ v,\ Q) & = & Q\ \wedge\ \exists p, s.\ \textbf{is\_mem}(p, s)\ \wedge\ p \leq m < p + s - (k-1) \\
\mathrm{WP}(\textbf{load}\ k\ r\ m,\ Q) & = & \forall v.\ Q[r \leftarrow v]\ \wedge \\
 & & \exists p, s.\ \textbf{is\_mem}(p, s) \wedge\ p \leq m < p + s - (k-1)
\end{array}
$$

5.2 VC generation

The next part of the puzzle is to compute post-conditions from block continuations. We start by defining rules for computing postconditions given some known assignment of preconditions to other labels, Γ:

$$
\begin{array}{lcl}
\mathrm{Post}(\texttt{exit}) & = & \Gamma_Q \\
\mathrm{Post}(\texttt{ja}\ l) & = & \Gamma(l) \\
\mathrm{Post}(\texttt{jcc}\ cc\ r\ v\ l_\mathrm{T}\ l_\mathrm{F}) & = & Q\ \texttt{\&\&}\ \Gamma(l_\mathrm{T}) \vee \neg Q\ \texttt{\&\&}\ \Gamma(L_\mathrm{F}) \\
 & & \text{where } Q = cc(r, v)
\end{array}
$$

Building on top of that, the pre-condition of an individual block can be acquired with:

$$\mathrm{Pre}(\textbf{block}\ l\ s\ j) = \mathrm{WP}(s, \mathrm{Post}(j))$$

Next, we need to solve the task of computing pre-conditions for every block, including assigning conditions to labels in Γ. If the control graph was acyclic, we could just just let the pre-condition for a label correspond to the WP result of the corresponding block. Knowing that the module was acyclic, we could then infer that all mappings $\Gamma(l)$ would be expanded without infinite recursion. If we let β be the collection of blocks, the label mapping could be expressed through the equality:

$$\Gamma(l) = \mathrm{Pre}(\beta(l))$$

We need another definition if the graph isn't acyclic however. If we were to use the same method, we would encounter recursive dependencies in the resulting in an infinitely long definition of some pre-conditions. In order to break such dependency cycles, we need to use a weaker condition $\Gamma(l) = R$ such that $R \implies \mathrm{Pre}(\beta(l))$ for some block l that occurs in the cycle. Crucially, R should itself occur as a part of $\mathrm{Pre}(\beta(l))$. Rather than attempting to automatically generate invariants, we allow the user to optionally annotate blocks with requirements, and add a well-formedness requirement that modules with cyclic control flow are annotated as to break all dependency cycles. If we let α denote these user annotations, the updated definition can be expressed as:

$$\Gamma(l) = \begin{cases} \mathrm{Pre}(\beta(l)) & \text{if } l \notin \alpha \\ R & \text{if } \alpha(l) = R \text{ and } \forall \textbf{r0} \ldots \textbf{r10}.\ R \implies \mathrm{Pre}(\beta(l)) \end{cases}$$

Note that this definition produces side conditions when encountering an annotated requirement.

As the last piece, we define the VC-gen for an entire module:

$$\text{VcGen}(\textbf{module: start } l \textbf{ blocks } \beta \textbf{ end}) = \forall \mathbf{r0} \ldots \mathbf{r10}.\ \Gamma_P \implies \text{Pre}(\beta(l))$$

6 Total correctness

Showing partial correctness for eBPF modules is useful, but that isn't enough to satisfy all requirements of the safety policy. Modules need to terminate, so we must prove total correctness if we wish to properly support modules with cyclic control flow.

While proving termination can be tricky, it is generally straightforward to generate verification conditions for a *structured* program with annotated invariants and variants. What complicates this task is that eBPF modules do not use structured control flow, instead allowing jumps between arbitrary positions in the code. This section details a proposal for how termination could be proven for a significant subset of eBPF modules.

Proving termination for general control graphs is a difficult task. However, the vast majority of eBPF modules are compiled from source code written in higher-level languages, most of which *do* follow structured flow. If we define similar well-formedness requirements for control graphs, we can limit the scope of termination proofs to make it a much more achievable goal.

6.1 Graph restrictions

We view each module as a directed control graph G with edges E and vertices V, of which one vertex v_{start} is the entry point of programs. We also define the *straight path* to a vertex v as a valid acyclic path $(v_{start} \to u_1 \to u_2 \to \cdots \to v) \in E$. The well-formedness requirement that we propose states that, aside from every vertex being reachable from the entry, every edge $v \to u \in E$ must be classifiable into one of two groups:

- *Forward edge*: No straight path to v contains u.
- *Backward edge (backedge)*: All straight paths to v contain u.

What this requirement essentially enforces is a chronological partial ordering of blocks. During execution, every time control is passed on to a new block, that block has either never been visited previously or it always has. This makes it much easier to reason about control flow, and we can use this to prove that our control graphs terminate.

Let $F \in E$ denote the set of forward edges while $B \in E$ denotes the set of backward edges, such that $F \cap B = \emptyset \wedge F \cup B = G$. If we exclude backward edges from the graph, the remaining edges F form a DAG. This is easily proven by contradiction: If there was some cycle $C = (v \to \ldots \to u \to v) \in F$, there would exist a straight path $(v_{start} \to \ldots \to v \to \ldots \to u) \in F$, and thus the edge $u \to v$ could not be classified as a forward edge in F, meaning that the cycle $C \in F$ cannot exist. From this we can also infer that a program excluding the backedges will terminate. Conversely, it could be shown that all backedges in B complete a cycle in F.

6.2 Proving termination

We prove termination by starting with an initial graph of all forward vertices and incrementally adding backedges to the graph while preserving termination.

First, we add some additional requirements. Every node that is the target of a backedge $v \to u$ must have one well-founded variant $\delta(u)$ associated with it. For every backedge in the graph $v \to u \in B$, it must be proven that following any path from u to v strictly decreases x. Thus when that backedge is traversed and control is passed to u again, this variant is strictly lower than the last time u was visited. It's important that the variant is bound to the backedge targets, not the backedges themselves. This ensures that all backedges to one node are proven to decrease the exact same variant.

Putting aside how this is proven for now, let us see how this allows us to prove termination for the entire graph. First we define a partial ordering for precedence of vertices: we say that v precedes u if there is a path from v to u in F, (empty paths not counted).

Base case: We start with the graph of forward edges, $G_{acc} = (V, F)$. Then we pick a backedge $v \to u \in B$ such that no other edge in B targets a vertex that precedes u. We remove it from B, and add it to G_{acc}. Assuming that a well-founded variant $\delta(u)$ has decreased on all paths from u to v, the new G_{acc} will still terminate since the backedge can only be reached a finite number of times.

Inductive step: Now we pick a new backedge $v' \to u'$ from B in the same manner as we did the first one. This is also removed from B and inserted into G_{acc}. Make the following observations:

- For all backedge targets $s \to t \in G_{acc}$, t may precede v', but v' cannot precede t.
- Thus the backedge $v' \to u'$ cannot be used to reach the targets of any other backedges, and cannot touch their variants either.
- The previously added backedges can *maybe* be used to reach u' without decreasing its corresponding variant. However, this can only happen a finite number of times, since all previous backedges terminate.

By induction, we can keep adding backedges in order of target precedence while preserving termination until $G_{acc} = G$, showing that G terminates.

6.3 Proof rules & VC-gen

Now, we return to the question of how we may track variants.

First, we extend our proof system and VC-gen with new rules. In addition to a mapping of associated requirements $\alpha : \mathit{label} \to \mathit{form}$, we require a user-annotated mapping of associated variants $\delta : \mathit{label} \to \mathit{expr}$. Every block that is a target of a back edge must have such a variant.

To simplify things, we limit our variants to expressions generated by *expr*, ensuring that it's well-founded by requiring that it is non-negative. It's necessary to create a new function for computing requirements for a jump, requiring a decrease in a variant based on whether the jump is backwards or not.

$$\mathrm{Jump}(l_s, l_d) = \begin{cases} \Gamma(l_d) & \text{if } l_d \notin \delta \text{ and } l_s \to l_d \in F \\ \Gamma(l_d) \wedge 0 \leq \delta(l_d) < x & \text{if } l_d \in \delta \text{ and } l_s \to l_d \in B \end{cases}$$

Here, x should be a fresh free variable, identified by the label l_d. This variable should only be bound by the function that defines $\Gamma(l)$. Next, we'll update our post-condition function to use the new jump function:

$$\begin{array}{lcl} \mathrm{Post}(l_s,\ \texttt{exit}) & = & \Gamma_Q \\ \mathrm{Post}(l_s,\ \texttt{ja}\ l_d) & = & \mathrm{Jump}(l_s,\ l_d) \\ \mathrm{Post}(l_s,\ \texttt{jcc}\ cc\ r\ v\ l_\mathtt{T}\ l_\mathtt{F}) & = & Q\ \texttt{\&\&}\ \mathrm{Jump}(l_s,\ l_\mathtt{T}) \vee \neg Q\ \texttt{\&\&}\ \mathrm{Jump}(l_s,\ L_\mathtt{F}) \\ & & \text{where } Q = cc(r, v) \end{array}$$

We also need to pass the block label to the post-condition function:

$$\mathrm{Pre}(\textbf{block}\ l\ s\ j) = \mathrm{WP}(s,\ \mathrm{Post}(l, j))$$

The result is that every backwards jump generates an additional requirement that the variant exprresion associated with the target block must be non-negative and strictly smaller than some free variable x. Now we need to bind the free variables that have been generated, so they represent the values of the variants at the beginning of the targeted blocks.

Since every path to the jumping point will include the target block l_d, we can bind the variable x when computing $\Gamma(l_d)$, assigning it to the expression $\delta(l_d)$. However, if any block from the target to the source has an associated requirement in α, this too will have to bind x. The associated requirement must then track the necessary information about the variant. To simplify things, let's assume that all backtargets and only backtargets have associated requirements. We then rework our definition $\Gamma(l)$ to bind x as well as all variables $x_{t1} \ldots x_{t2}$ that $\beta(l)$ must track:

$$\Gamma(l) = \begin{cases} \mathrm{Pre}(\beta(l)) & \text{if } \beta(l) \text{ is not a backtarget} \\ R & \text{if } \alpha(l) = R \text{ and } \forall \mathbf{r0} \ldots \mathbf{r10},\ x, x_{t1} \ldots x_{t2}.\ R \wedge v = \delta(l) \implies \mathrm{Pre}(\beta(l)) \end{cases}$$

6.4 Well-formedness verification

A challenge to this approach is that we must classify all edges and verify that the control graph actually follows our new well-formedness requirement. While a simple path enumeration algorithm could correctly perform the check, it would have poor runtime performance. We know which nodes should be the targets of backward jumps from annotations, and we could potentially require annotations of backwards jumps. This might make it the pre-requisites easier to verify.

6.5 Expressiveness & practicality

So how well does the system work in practice? Which programs can be proven with it, and how much extra work is required by both users and provers?

The restrictions placed on the control graph are loose enough that structured programs can be translated directly to valid control graphs. However, we aren't accounting for the kinds of transformations that an optimizing compiler might apply. For example, if a compiler were to generate a cycle with several entrypoints, that would be an invalid program.

7 Related work

7.1 PREVAIL

PREVAIL [5] is an eBPF verifier, developed as an alternative to the Linux verifier. The project claims a number of advantages, among them: faster runtime performance, greater precision in modeling of register values, and support for loops.

The paper identifies four main issues that eBPF developers struggle with: a high amount of false positives, bad scaling to programs with many paths, a lack of support for loops (at the time), and a lack of formal foundation. The solution to these problems is given by a formalized semantics for eBPF and a verifier based on abstract interpretation. This verifier is supposed to scale better with program complexity and support a wider range of programs resulting in fewer false positives. One claim of the paper is that their verifier supports loops, but at the time the paper was written, this support did not include validating termination. It seems that some support for termination analysis has since been added, but the details remain undocumented.

The paper defines a concrete semantics for a large subset of the eBPF language, the operational semantics defined such that the machine aborts on any unsafe operation. The task of the verifier is then to ensure that an eBPFPL program never aborts. The analysis uses abstract interpretation, adapting the concrete semantics to an abstract semantics based on domains for abstract numerical values and bounded sets. In contrast to the non-relational interval tracking of the Linux verifier, PREVAIL is able to track relations between registers using inequalities of the form $r_a - r_b \leq C$.

The combination of abstract interpretation and a good abstract domain does allow PREVAIL to verify a wider range of programs. However, an important question to ask is whether these expanded capabilities are enough. Are there use cases where the verifier falls short? An argument is made in the abstract of the paper:

> Our choice of abstraction is based on common patterns found in many eBPF programs. We observed that eBPF programs manipulate memory in a rather disciplined way which permits analyzing them successfully with [methods we employ in our analyzer.]

In other words, the breadth of the patterns and features that PREVAIL supports is informed by common patterns observed in real-world eBPF modules. An underlying assumption is made here, that an empirical analysis of current eBPF usage, along with current pain points and their workarounds, can reveal a *stable* model of the applications of and requirements for eBPF. However, it has to be considered that observed real-world eBPF programs are restricted and limited by the current capabilities and limitations of the kernel subsystem and its verifier. Ever since it's inception as a very limited DSL, the now JIT-compiled verified virtual machine — that we refer to as "*Extended Berkeley Packet Filters*" — has been expanding in scope. If the boundaries of valid programs are expanded, developers may very well expand their expectations in a case of induced demand.

In comparison, a PCC architecture can reach a point where it is more or less "done". It is much easier to reach a point of near completeness, where most programs that are safe can be proven safe. At such a point, the verification architecture only needs to be extended when new functionality is added to the system itself in the form of new instructions, helper functions, and module types. Even then, extensions will generally be limited to the addition of new rules.

That being said, proof carrying code and static analysis aren't necessarily incompatible approaches. At the end of the day, the job of a static analyzer is to convince itself that a program does indeed follow a safety policy. If the analyzer is sound, accepting a program implies that a safety proof exists, and the safety of the program has been established based on a program model explored by the analysis. It is a bit of a leap, but it might be possible for the traces of analysis to be used for generating safety proofs. In such a scenario, a PCC system might end up serving a role that can be more accurately be described as a safe interface between userspace analyzers and the kernel.

8 Implementation

To test the VC-gen described in section 5, we create a proof-of-concept implementation (*only for the partial correctness VC-gen*). The implementation is a relatively straightforward pipeline:

1. A module written in annotated assembly is taken as input.
2. The module is parsed into a line-by-line representation.
3. The parsed module is pre-processed into a control graph representation.
4. The control graph is fed to an implementation of the VC-gen algorithm.
5. The result of the VC-gen is exported to WhyML.
6. The verification conditions are passed to Why3 for automated solving.

8.1 Parsing & syntax

eBPF programs are written in a NASM-style assembly language, matching that of the **ebpf-tools**[6] assembler. In order to simplify internal processing, we require that jumps always target labels. In addition to writing instructions and comments, users can also write annotations. Specifically, modules can be annotated with requirements and ensurances, and labels can be annotated with requirements. Annotations are prefixed with `;#` and are written in a custom formula language.

In the implementation, internal custom data types are used to represent expressions, formulas, and eBPF programs. This makes it easier to both parse from and export to several different formats if needed. Several helper functions are defined for concisely constructing and modifying existing formulas.

8.2 Pre-processing

In the pre-processing step, the input assembly program is transformed into a control graph. The creation of nodes is accomplished by splitting the line sequence at every label, continuation, and continuation target. Nodes are referenced by labels, with new labels being generated for nodes that don't have one. This step also checks the module for well-formedness with respect to labels and jumps, ensuring that all continuations are valid.

8.3 VC-gen

Next, VC generation is performed on the control graph. Dependencies between blocks are resolved by traversing the control graph in topological order. This ensures that every block is processed before any other block that depends on its precondition. If a program hasn't been annotated to break up all circular dependencies, this is detected and reported as an error.

Each block is processed as follows (where Γ is a store of preconditions):

1. Generate a post-condition from continuation, looking up preconditions in Γ.
2. Run the weakest precondition calculus on the postcondition with the block body. The WP-calculus is a more or less direct translation of the formal specification.
3. If the block has an annotated requirement, generate a side condition. Otherwise, save the new precondition to Γ.

8.4 Exporting to a solver

At this point we have our verification conditions, represented in an internal format. Our next step is to export these to a language that a solver or proof framework can understand.

Our export language of choice is WhyML[2], the primary language used by the Why3 framework. This choice is not founded on strong convictions, but Why3 does have some advantages. It's a system for

interactive proving of programs, and as such is already oriented towards the task at hand. Furthermore, WhyML can act as an intermediate representation which Why3 can parse and delegate to a wider range of solvers and provers than an interface such as SMTLIBv2 might support. This isn't to say that the WhyML language would function well as a kernel interface. It's a very high level language, and Why3 has no accessible support for proof extraction. Rather, it's a quick solution for simply checking validity of the verification conditions we generate.

9 Experiments

In this section, various synthetic eBPF modules are tested against our own VC solution and the linux verifier. These tests are constructed to shed some light on the capabilities of our solution, its shortcomings, and these in comparison to the current eBPF architecture.

It isn't fully possible to compare the two on an even playing field. The VC-gen supports a small subset of the instruction set, so most real-world programs will simply not parse. Additionally, several safety requirements aren't checked by the VC-gen at all. These shortcomings are a result of prioritization, and as such aren't all that interesting. We are more interested in uncovering shortcomings in areas that *have* been prioritized. Just as big a hindrance to comparison are the differences in the *feature support* of the two systems. One of our goals is to prevent division by zero, while the eBPF subsystem will happily let $x/0 = 0$. Similarly, our PCC architecture paves the way for dynamically sized arrays, but the verifier lacks the ability to represent array size as anything other than a static number. As a result of these problems, both systems will be compared in the ways that they are comparable, and the VC-gen architecture should be examined on its own when comparison isn't possible.

9.1 Setup

9.1.1 Prelude

`ebpf-vc` is designed so that pre- and postconditions can be annotated as requirements in the header of the an assembly module. Getting the same set of pre-conditions in real eBPF isn't quite as straightforward. In order to simplify testing, a general tester program `verifier.c` is written. This will interact with the BPF subsystem in order to correctly load modules and verify them. Taking inspiration from [13], a prelude is prepended to every eBPF module. This prelude uses maps to initialize a few registers with meaningful values before execution of the program text:

- `r1`: Pointer to a buffer of size 64.
- `r2-r4`: Arbitrary integers.

This gives us an initial configuration to play around with for our experiments.

9.1.2 Reproducing experiments

The experiments were carried out using the following software:

- VC-gen:
  - The `ebpf-vc` software developed in this project.
  - The Why3 platform w/ CVC4 1.8.
- Linux verifier:
  - GCC version 11.3.0.
  - A machine running Linux 5.15.0 with eBPF features enabled.
  - `ebpf-tools`[6] (forked to support labels).

`ebpf-vc` can be built and run using `cargo`. It takes a single assembly file and generates verification conditions in WhyML. These can then be passed to `why3` for solving, using CVC4 as a backend:

```
1    cargo run --release samples/gcd.asm > gcd.mlw;
2    why3 --debug=ignore_unused_vars prove -p CVC4,1.8 gcd.mlw;
```

This should produce a file `gcd.asm` with the following contents:

```
use mach.int.UInt64
use int.Int
use int.ComputerDivision
predicate is_buffer (p: uint64) (s: uint64)

goal loop: forall r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 : uint64 . ((r2 <> 0) -> (forall v4 : uint64 .
↪  ((v4 = r1) -> ((r2 <> 0) && (forall v3 : uint64 . ((v3 = mod v4 r2) -> (forall v1 : uint64 . ((v1
↪  = v3) -> (((v1 <> 0) && (v1 <> 0)) \/ (not ((v1 <> 0)) && true)))))))))

goal entry: forall r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 : uint64 . (true -> (((r2 = 0) && true) \/ (not
↪  ((r2 = 0)) && (r2 <> 0))))
```

And Why3 should print the following to the terminal:

```
File "test.mlw", line 6, characters 11-167:
Goal loop.
Prover result is Valid (0.05, 2203 steps).

File "test.mlw", line 8, characters 12-78:
Goal entry.
Prover result is Valid (0.05, 2203 steps).
```

The verifier can be accessed using `verifier.c`, which bundles up an eBPF module with the prelude and then attempts to load it, reporting whether the verifier accepted or rejected the module. eBPF modules must be given in binary form, so it is necessary to compile them using `ebpf-tools`.

```
    ebpf-tools -a -o gcd.bin gcd.asm;
    gcc -o verifier extra/verifier.c;
    ./verifier gcd.bin;
    ./verifier gcd.bin --log;
```

`ebpf-tools` should produce a binary file `gcd.bin`, and `verifier` should either print `Program accepted` or `Program denied` along with some extra information.

9.2 Samples

The following table summarizes the results, showing whether the verifier and the VC-gen correctly determine whether each given module is safe. The last column lists special circumstances for either tools.

Experiment	Kernel(1)	`ebpv-vc`(2)	Remarks
`divzero_indirect`	✓	✓	1: $\mathbf{div}(x, 0) = 0$
`mem_indirect.asm`	✗	✓	
`dag.asm`	✓	✓	
`simple_loop.asm`	✓	✓	2: Partial correctness
`simple_loop_bad.asm`	✓	✗	2: Partial correctness
`weird_loop.asm`	✗	✓	2: Partial correctness
`sum.asm`	✗	✓	1: Not typable
`sum64.asm`	✓	✓	2: Partial correctness
`partition.asm`	✗	✓	1: Not typable 2: Partial correctness
`save_to_frame.asm`	✓	✗	

9.2.1 Division by zero

While the eBPF subsystem has a workaround for the problem of potential zero-divisions, it's still a valuable to find a more robust solution. We cannot make the kernel verifier reject potential divisions by zero. However, we *can* inspect the verifier logs in order to determine its knowledge about the divisor at the time of division. Based on this, we can infer whether or not the verifier would be able to accept divisions as safe. In fig. 6, the knowledge that $\mathbf{sub}(\mathbf{r2}, \mathbf{r3}) = 0$ must be inferred from the fact that they are non-equal. The kernel verifier is not able to do this.

```
1      mov r0 0
2      jeq r2 r3 skip  ; Ensure that r2 <> r3
3      mov r1 r2
4      sub r1 r3
5      mov r0 r4
6      div r0 r1       ; We indirectly know that r2 - r3 <> 0
7  skip:
8      exit
```

Figure 6: `divzero_indirect.asm` — Division by a non-zero register.

9.2.2 Indirect relations between registers

In general, the VC-gen is good at tracking the relations between registers, rather than tracking them individually. In fig. 7, an upper bound on **r6** can be inferred through a condition on **r4**. The kernel verifier isn't able to make that connection once **r6** has been modified from the original value of **r4**.

```
1  ;# requires is_buffer(r1, r2)
2  ;# requires r2 = 64
3      mov r0 0
4      jge r4 1000 skip   ; avoid overflows
5      mov r6 r4
6      mul r6 2
7      jge r4 32 skip     ; r2 < 32 => r6 < 64
8      mov r5 r1
9      add r5 r6
10     ldxb r0 [r5]
11 skip:
12     exit
```

Figure 7: `mem_indirect.asm` — Memory access indirectly known to be safe.

9.2.3 Acyclic control flow

Needing to annotate individual blocks with requirements is tedious. However, this is only necessary when the control graph contains loops. The VC-gen works without annotations as long as the module is acyclic, as exemplified in fig. 8.

```
;# ensures r0 >= r2
;# ensures r0 >= r3
;# ensures r0 >= r4
a:
    jlt r2 r3 c
b:
    jge r2 r4 R2
c:
    jge r3 r4 R3
    ja R4
R2:
    mov r0 r2
    ja end
R3:
    mov r0 r3
    ja end
R4:
    mov r0 r4
end:
    exit
```

Figure 8: `dag.asm` — Computes the maximum of three registers.

9.2.4 Loops

This brings us to loops. The verifier does support validating bounded loops to a limited extent. If the feature is enabled, it is generally necessary to load modules as root to access the functionality. The VC-gen implementation isn't capable of showing total correctness, so it can only be evaluated on whether it validates partial correctness.

It is possible to verify simple loops such as the one in fig. 9. The kernel will also reject an infinite loop such as the one shown in fig. 10, but it does so by exhausting the limit of total states. Surprisingly however, the verifier does fail on some very simple loops. In fig. 11, the value of **r2** is not precisely known at runtime, but it is limited to a maximum of 1000. This indirection is enough for verifier to reject the program, despite that the module terminates within a reasonable bound. It seems that the support for loops is intended for specific, simple use cases.

```
    mov r0 0
    mov r1 0
    mov r2 32
loop:
    add r0 r1
    add r1 1
    jlt r1 r2 loop
    exit
```

Figure 9: `simple_loop.asm` — Computes a triangular sum through a loop.

```
    mov r0 0
    mov r1 0
    mov r2 32
loop:
    add r0 r1
    add r1 1
    jeq r1 r1 loop
    exit
```

Figure 10: `simple_loop_bad.asm` — Infinite loop.

```
    jgt r2 1000 end
    mov r3 0
loop:
    add r3 1
    jlt r3 r2 loop
end:
    exit
```

Figure 11: `weird_loop.asm` — Terminating loop with a non-register variant.

9.2.5 Dynamic buffers

One thing that the kernel verifier doesn't support is dynamically sized buffers. It isn't quite right to say that the verifier won't validate fig. 12, rather, it simply isn't possible to correctly submit the program to the verifier in the first place. However, there are no issues validating the alternative statically sized version in fig. 13.

```
;# requires is_buffer(r1, r2)
;# requires r2 > 1
    mov r3, 0 ; idx
    mov r0, 0
loop:
;# req is_buffer(r1, r2)
;# req r3 < sub(r2, 1)
    mov r4 r1
    add r4 r3
    ldxh r4 [r4]
    add r0 r4       ; load element and add to sum
    add r3 2
    mov r6 r2
    sub r6 1
    jlt r3 r6 loop ; loop if address is 2 lower than size
    mov r0 r0
    exit
```

Figure 12: `sum.asm` — Sum all halfwords in a dynamically sized buffer.

```
;# requires is_buffer(r1, r2)
;# requires r2 = 64
    mov r3, 0 ; idx
    mov r0, 0
loop:
;# req is_buffer(r1, r2)
;# req r2 = 64
;# req r3 < sub(r2, 1)
    mov r4 r1
    add r4 r3
    ldxh r4 [r4]
    add r0 r4      ; load element and add to sum
    add r3 2
    jlt r3 63 loop ; loop if address is 2 lower than size
    mov r0 r0
    exit
```

Figure 13: `sum64.asm` — Sum all halfwords in a fixed-sized buffer.

While it isn't possible for the VC-gen to validate termination of the `sum` programs, it can correctly model loop invariants to check that array access is performed correctly. It does the same for fig. 14, a larger array transformation that partitions elements based on whether they equal zero or not.

```
;# requires is_buffer(r1, r2)
;# requires r2 > 0
    mov r3 0            ; i = 0
    mov r4 r2
    sub r4 1            ; j = len array - 1
loop:
;# req is_buffer(r1, r2)
;# req r4 < r2
;( req 0 <= r3 is needed for termination)
    jle r4 r3 return    ; while i < j
caseA:
    mov r5 r1
    add r5 r3
    ldxb r5 [r5]
    jne r5 0 caseB      ; if a[i] <> 0 goto B;
    add r3 1            ; i += 1; continue;
    ja continue
caseB:
    mov r6 r1
    add r6 r4
    ldxb r6 [r6]
    jeq r6 0 caseC      ; if a[j] == 0 goto C;
    sub r4 1            ; j -= 1; continue;
    ja continue
caseC:
    mov r7 r1
    add r7 r3
    stxb [r7] r6
    mov r7 r1
    add r7 r4
    stxb [r7] r5        ; a[i] <-> a[j]
; end
continue:
    ja loop
return:
    exit
; done
```

Figure 14: `partition.asm` — Partition an array into zero and non-zero sections.

9.2.6 Memory modeling

Another quite severe limitation of the current VC-gen is the black-box approach to memory. It will fail to verify a program like fig. 15 that stores a pointer in the stack frame before using it. In comparison, the verifier does track the contents of the of the stack frame and verifies this just fine.

```
;# requires is_buffer(r1, r2)
;# requires r2 = 8
;# requires is_buffer(r10, r9)
;# requires r9 = 512
stxdw [r10 + -8] r1
ldxdw r1 [r10 + -8] ; erases knowledge of r1
ldxdw r3 [r1]
exit
```

Figure 15: `save_to_frame.asm` — Saving a pointer to the stack, retrieving it, and dereferencing it.

9.3 Integer overflow

During experimentation, an error was discovered in the VC-gen implementation. Automatic type-casting in WhyML results in a bug where several bitstring values are converted back into unbounded integers for certain operations. The consequence is that integer overflows are not detected correctly, potentially resulting in validation of unsafe modules. This is a good reminder that proof systems can be the subject of logic errors just as any other system. However, this can mostly be chalked up to an eager cast in a high-level proof framework. A real PCC architecture would define an independent fully specified logic for working with bit strings, that would reject any proof that relies on such casting. Some amount of errors is going to be unavoidable, but this error shouldn't occur in kernel space.

10 Evaluation

The results from the experiments are promising overall. It was possible to show several instances in which automated PCC using VC-gen and a solver allows for the verification of valid eBPF modules that the verifier is unable to accept. There were some issues regarding the generation of WhyML and using bitstrings, but these can be corrected, either through patching the WhyML exporter or choosing a different backend. The experience of proving safety through VC-gen was overall pleasant as well. For the most part, programs didn't need annotations in order for the validation to go through.

When loops are involved, the necessary invariant annotations can be placed in positions corresponding roughly to similar locations where invariants are annotated in high-level proof languages. This is one point where the kernel verifier alternatives like PREVAIL have some advantage in immediate ergonomics. No annotations are required to ensure safety for the loops that these verifiers support. However, as shown in the experiments, the loop support of the kernel verifier is very limited. Regrettably, termination proofs weren't implemented as a proof of concept in the scope of this project. It would have been interesting to see how well the proposed VC-gen for total correctness would function in practice. Cyclic control flow is one of the areas where PCC could drastically improve eBPF ergonomics compared to the current state of affairs.

There were a significant amount of safety policies that aren't enforced by `ebpf-vc`. Some, like memory alignment, are relatively simple to add, while others, like leak prevention, may be significantly more difficult. Barely mentioned was the prevention of side-channel leaks through speculative execution, and this is major surface area which has already been the target of malicious attacks several times. That is to say, this VC-gen only demonstrates viability in certain areas, although the results look promising.

When it comes to alternatives such as PREVAIL, the choice is essentially between replacing the current technology with a somewhat incremental improvement or adopting a fundamentally different approach. With the work required in order to make such a switch, one might argue that switching to a "fuller" solution would be more future-proof than choosing one that seems "good enough" for the time being. Additionally there isn't any immediate reason that an automatic proof generator wouldn't be able to validate the programs that static analysis engines can.

All in all, we have demonstrated PCC as a potential architecture for verifying eBPF, focusing mainly on the aspects of conditional branching and cyclic control flows.

10.1 Design decisions

One of the major decisions that I made in this project was to limit the breadth of support for different parts of eBPF, and instead focus mainly on conditional jumps and conditional branching. This prioritization worked out well, as there was more than enough challenges to tackle in this area alone. This did have the downside that experimentation on real-world eBPF modules wasn't viable, as nearly all real eBPF applications use features outside of our scope. However, I have produced a proof of concept for a partial correctness VC-gen as well as a proposal for total correctness VC-gen for cyclic control graphs. That I consider a success.

In terms of designing the logic and VC-gen, the major choice I made was to deal with circular dependencies by treating preconditions as auxiliary mappings and requiring annotations of invariants, which turned out to be a good solution. If the inference rules were to be further developed, I would explore whether they could be more concisely stated by making the collection of blocks auxiliary as well.

As for the choices of supporting technologies, things generally went well. Programming the PoC implementation in Rust worked out smoothly. The strong type system allowed me to use parser combinators for my input processing, while the imperative coding paradigm made it easy to implement a dependency resolution algorithm. Exporting to Why3 allowed me to quickly get a working proof of concept up and running. However, this choice also had some unintended consequences. Why3 doesn't support proof extraction, but it was hard to justify writing a new backend for another solving, the

result being that the proof-checking step is absent from the final PoC. Additionally, what seems to be an automatic type-casting rule in WhyML led to errors in the generated proof conditions.

11 Conclusion

11.1 Summary

The eBPF subsystem for Linux is an emerging technology used for kernel extensions that poses challenges to the safety and security of Linux. The properties of eBPF make it an interesting target for code validation using proof-carrying code. In this project, we have explored the eBPF technology and identified safety requirements that eBPF programs must uphold to run in kernel space. A logic and verification condition generator has been developed for validating a subset of those requirements, focusing on branching and cylic control flow. A partial proof-of-concept PCC solution has been developed, implementing a VC-gen and dispatching proof obligations to a solver. This has been compared to the current in-kernel verifier using a small collection of synthetic programs. The PCC solution has been shown to correctly validate some programs which the current verifier cannot. In addition to the PCC demonstration that validates partial correctness, a solution for validating termination of looping eBPF programs has been proposed. All in all, this shows promising results with regards to PCC for eBPF, with much work yet to be done.

11.2 Future work

There are many more areas to explore before a proper PCC for eBPF solution can be developed. Listed below are some of the tasks that would be required, though several of these may have been covered by other projects to varying degrees.

- Defining a formalized semantics that plays nicely with the inference rules.
- Defining a proper model of bit strings.
- Refining inference rules, VC-gen, and proving soundness.
- Properly exploring approaches to proving termination.
- Handling remaining eBPF instructions:
  - 32-bit arithmetic operations.
  - Calling helper functions.
  - Memory beyond the black-box approach.
  - Atomic operations.
  - Specialized instructions (**LoadMapFd**, maybe more?).
- Handling remaining safety policies:
  - Memory alignment.
  - Secret leakage prevention.
  - Side-channel attack prevention.
- Designing inference rules for proving verification conditions.
- Designing a proof delivery format.
- Implementing a proof of concept in-kernel eBPF subsystem using PCC.

References

[1] Bpf documentation. Accessed Mar 14, 2023.

[2] The why3 platform. Accessed Mar 15, 2023.

[3] Paul Chaignon. Complexity of the bpf verifier, 2019. Accessed Mar 8, 2023.

[4] ebpf.io. Accessed Feb 2, 2023.

[5] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Ken Friis Larsen. ebpf-tools, 2023. Accessed Mar 3, 2023.

[7] Linux User's Manual. bpf(2), bpf-helpers(7), 2023.

[8] Claude Marché. Lecture notes in mpri course 2-36-1: Proof of program, 2013.

[9] T.J. McCabe. A complexity measure. *IEEE transactions on software engineering*, SE-2(4):308–320, 1976.

[10] GREG MORRISETT, KARL CRARY, NEAL GLEW, and DAVID WALKER. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, 2002.

[11] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[12] Marta Rybczyńska. Bounded loops in bpf for the 5.3 kernel, 2019.

[13] Mads Obitsøe Thomsen. ebpf and pcc. Master's thesis, University of Copenhagen, 2022.

[14] Terry Yin. Lizard, 2023. Accessed Mar 8, 2023.

A ebpf-vc - Proof of concept VC-generator

All code is available at `https://github.com/korreman/ebpf-vc`.
The listings reflect commit `82d96c3`.

A.1 src/main.rs

```rust
use argh::FromArgs;

use std::{ffi::OsString, process::ExitCode, str::FromStr};

use ebpf_vc::{
    cfg::{Cfg, ConvertErr},
    formula::FormulaBuilder,
    parse::module,
    vc::vc,
    whyml,
};

#[derive(FromArgs)]
/// A verification condition generator for eBPF.
struct EbpfVc {
    /// input to generate conditions for
    #[argh(positional)]
    file: OsString,
    /// proof obligation format (default is WhyML)
    #[argh(option, default = "OutputFmt::WhyML")]
    format: OutputFmt,
}

enum OutputFmt {
    WhyML,
    CVC5,
}

impl FromStr for OutputFmt {
    type Err = &'static str;
    fn from_str(s: &str) -> Result<Self, Self::Err> {
        let fmt = match s.to_ascii_lowercase().as_str() {
            "whyml" => Self::WhyML,
            "cvc5" => Self::CVC5,
            _ => return Err("unknown output format"),
        };
        Ok(fmt)
    }
}

fn main() -> ExitCode {
    let opts: EbpfVc = argh::from_env();

    let file = std::fs::read_to_string(opts.file);
    let contents = match file {
        Ok(c) => c,
        Err(e) => {
```

```
            eprintln!("error: {e}");
            return ExitCode::FAILURE;
        }
    };

    let parsed_file = module(contents.as_str());
    let ast = match parsed_file {
        Ok((_, a)) => a,
        Err(e) => {
            eprintln!("error: failed to parse module - {e}");
            return ExitCode::FAILURE;
        }
    };
    //eprintln!("{ast:#?}\n");

    let mut f = FormulaBuilder::new();
    let preprocess_res: Result<Cfg, ConvertErr> = Cfg::create(ast, &mut f);
    let processed_ast = match preprocess_res {
        Ok(p) => p,
        Err(e) => {
            eprintln!("error: {e}");
            return ExitCode::FAILURE;
        }
    };
    //eprintln!("{processed_ast:#?}\n");

    let vc_res = vc(processed_ast, &mut f);
    match opts.format {
        OutputFmt::WhyML => println!("{}", whyml::Conditions(vc_res)),
        OutputFmt::CVC5 => eprintln!("Architecture currently cannot support both formats"),
    }
    ExitCode::SUCCESS
}
```

A.2 src/vc.rs

```
//! Verification condition generation.

use std::collections::HashMap;

use crate::{cfg::*, formula::*};

#[derive(Debug, PartialEq, Eq)]
enum BlockStatus {
    Pending,
    Cyclic,
    PreCond(Formula),
}

pub fn vc(module: Cfg, f: &mut FormulaBuilder) -> Vec<(String, Formula)> {
    // Stores results.
```

```rust
    let mut verif_conds: Vec<(String, Formula)> = Vec::new();

    // Stores cached preconds for each block.
    // Also used to track which blocks have already been visited.
    let mut pre_conds: HashMap<Label, BlockStatus> = HashMap::new();

    // Perform a reverse breadth-first traversal of CFG.
    let mut stack = vec![module.start.clone()];
    while let Some(label) = stack.pop() {
        // Try to mark block as pending.
        let status = pre_conds
            .entry(label.clone())
            .or_insert(BlockStatus::Pending);
        // Skip block if it has already been processed.
        if matches!(*status, BlockStatus::PreCond(_)) {
            continue;
        }
        let block = &module.blocks[&label];

        // This func attemps to retrieve the precond of another block.
        let mut get_post_cond = |target: &Label| {
            let b = pre_conds.get_mut(target);
            match b {
                // If already processed, return the result.
                Some(BlockStatus::PreCond(c)) => Some(c.clone()),
                // If pending, use the requirement if it exists and fail if it doesn't.
                Some(BlockStatus::Pending) | Some(BlockStatus::Cyclic) => {
                    // Mark block as cyclic.
                    *b.unwrap() = BlockStatus::Cyclic;
                    if let Some(c) = &module.blocks[target].require {
                        Some(c.clone())
                    } else {
                        Some(f.top())
                    }
                }
                // If block isn't marked as anything, push the current block and it to the stack.
                None => {
                    stack.push(label.clone());
                    stack.push(target.clone());
                    None
                }
            }
        };

        // Generate postcond from the continuation of the block.
        let post_cond = match &block.next {
            Continuation::Exit => Some(module.ensures.clone()),
            Continuation::Jmp(target) => get_post_cond(target),
            Continuation::Jcc(cc, lhs, rhs, target_t, target_f) => {
                // First, get postcond of the two targets.
                let cond_t = get_post_cond(target_t);
                let cond_f = get_post_cond(target_f);

                // Next, build formula for comparison.
                let lhs = f.reg(*lhs).0;
```

```
                let rhs = match rhs {
                    RegImm::Reg(r) => f.reg(*r).0,
                    RegImm::Imm(i) => f.val(*i),
                };
                let cc = f.rel(*cc, lhs, rhs);

                // Generate condition as conjugation between the two branches.
                cond_t.zip(cond_f).map(|(cond_t, cond_f)| {
                    f.or(
                        f.asym_and(cc.clone(), cond_t),
                        f.asym_and(f.not(cc), cond_f),
                    )
                })
            }
        };
        // If the postcond couldn't be generated,
        // both the current block and any jump targets have been pushed to the stack.
        // We continue in order to handle new targets first.
        let post_cond = match post_cond {
            Some(c) => c,
            None => continue,
        };

        // Perform WP-calculus on postcond with block body.
        let wp_result = wp(f, &block.body, post_cond);

        // Cache or use result of WP.
        let top = f.top();
        let require = block.require.as_ref();
        let require = require.or(if pre_conds[&label] == BlockStatus::Cyclic {
            Some(&top)
        } else {
            None
        });

        if let Some(require) = require {
            // If the block has a requirement,
            // add a VC requiring that the requirement implies the WP result.
            verif_conds.push((label.clone(), f.implies(require.clone(), wp_result)));
            pre_conds.insert(label, BlockStatus::PreCond(require.clone()));
        } else {
            // Otherwise, cache the WP result for the block.
            pre_conds.insert(label, BlockStatus::PreCond(wp_result));
        }
    }

    // Add the precond of the starting block as a VC.
    verif_conds.push((
        "entry".to_owned(),
        match &pre_conds[&module.start] {
            BlockStatus::PreCond(c) => f.implies(module.requires, c.clone()),
            _ => panic!("starting block is never processed"),
        },
    ));
    verif_conds
```

```
}

fn wp(f: &mut FormulaBuilder, instrs: &[Stmt], mut cond: Formula) -> Formula {
    for instr in instrs.iter().rev() {
        match instr {
            Stmt::Unary(WordSize::B64, op, reg) => {
                let (t, t_id) = f.reg(*reg);
                let e = f.unop(*op, t);
                cond = assign(f, &t_id, e, cond);
            }
            Stmt::Binary(WordSize::B64, op, dst, src) => {
                let (d, d_id) = f.reg(*dst);
                let s = match src {
                    RegImm::Reg(r) => f.reg(*r).0,
                    RegImm::Imm(i) => f.val(*i),
                };
                let e = f.binop(*op, d, s.clone());
                cond = assign(f, &d_id, e, cond);

                // Add extra conditions for division/modulo by zero.
                if op == &BinAlu::Div || op == &BinAlu::Mod {
                    cond = f.asym_and(f.rel(Cc::Ne, s, f.val(0)), cond);
                }
            }
            Stmt::Store(size, mem_ref, _) => {
                let valid_addr = valid_addr(f, *size, mem_ref);
                cond = f.and(valid_addr, cond);
            }
            Stmt::Load(size, dst, mem_ref) => {
                let valid_addr = valid_addr(f, *size, mem_ref);
                let (_, v_id) = f.var(String::from("v"));
                let (_, t_id) = f.reg(*dst);
                let replace_reg = match f.replace(&t_id, &v_id, &cond) {
                    Some(x) => f.forall(v_id.clone(), x),
                    None => cond,
                };
                cond = f.and(valid_addr, replace_reg);
            }
            Stmt::Assert(a) => {
                cond = f.asym_and(a.clone(), cond);
            }
            instr => panic!("not implemented: {instr:?}"),
        }
    }
    cond
}

fn assign(f: &mut FormulaBuilder, target: &Ident, e: Expr, cond: Formula) -> Formula {
    let (v, v_id) = f.var(String::from("v"));
    match f.replace(target, &v_id, &cond) {
        Some(x) => f.forall(v_id.clone(), f.implies(f.eq(v, e), x)),
        None => cond,
    }
}

```

```rust
fn valid_addr(f: &mut FormulaBuilder, size: WordSize, MemRef(reg, offset): &MemRef) -> Formula {
    let (ptr, ptr_id) = f.var("p".to_owned());
    let (sz, sz_id) = f.var("s".to_owned());
    let addr = f.binop(BinAlu::Add, f.reg(*reg).0, f.val(*offset));
    let bytes = match size {
        WordSize::B8 => 1,
        WordSize::B16 => 2,
        WordSize::B32 => 4,
        WordSize::B64 => 8,
    };
    let upper_bound = f.binop(
        BinAlu::Sub,
        f.binop(BinAlu::Add, ptr.clone(), sz.clone()),
        f.val(bytes - 1),
    );
    f.exists(
        ptr_id.clone(),
        f.exists(
            sz_id,
            f.and(
                f.is_buffer(ptr_id, sz),
                //f.and(
                //    f.eq(f.binop(BinAlu::Mod, addr.clone(), f.val(bytes)), f.val(0)),
                f.and(
                    f.rel(Cc::Le, ptr, addr.clone()),
                    f.rel(Cc::Lt, addr, upper_bound),
                ),
                //),
            ),
        ),
    )
}
```

A.3 src/ast.rs

```rust
//! An AST used for parsing eBPF assembly.

#[rustfmt::skip]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum WordSize {
    B8, B16, B32, B64,
}

#[rustfmt::skip]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum Cc {
    Eq, Gt, Ge, Lt, Le, Set, Ne, Sgt, Sge, Slt, Sle,
}

#[rustfmt::skip]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
```

```rust
pub enum UnAlu {
    Neg, Le, Be,
}

#[rustfmt::skip]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum BinAlu {
    Mov, Add, Sub, Mul, Div, Mod, And, Or, Xor, Lsh, Rsh, Arsh,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct Reg(u8);
impl Reg {
    pub const R0: Self = Reg(0);
    pub const R1: Self = Reg(1);
    pub const R2: Self = Reg(2);
    pub const R3: Self = Reg(3);
    pub const R4: Self = Reg(4);
    pub const R5: Self = Reg(5);
    pub const R6: Self = Reg(6);
    pub const R7: Self = Reg(7);
    pub const R8: Self = Reg(8);
    pub const R9: Self = Reg(9);
    pub const R10: Self = Reg(10);

    pub fn new(id: u8) -> Option<Self> {
        if id < 11 {
            Some(Self(id))
        } else {
            None
        }
    }

    pub fn get(&self) -> u8 {
        self.0
    }
}

pub type Imm = i64;

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum RegImm {
    Reg(Reg),
    Imm(Imm),
}

pub type Offset = i64;
pub type Label = String;

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct MemRef(pub Reg, pub Offset);

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Stmt {
    Assert(Formula),
```

```
    Unary(WordSize, UnAlu, Reg),
    Binary(WordSize, BinAlu, Reg, RegImm),
    Store(WordSize, MemRef, RegImm),
    Load(WordSize, Reg, MemRef),
    LoadImm(Reg, Imm),
    LoadMapFd(Reg, Imm),
    Call(Imm),
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Cont {
    Jmp(Label),
    Jcc(Cc, Reg, RegImm, Label),
    Exit,
}

pub type Ident = String;

/// Expression used for formulas.
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Expr {
    Val(Imm),
    Var(Ident),
    Unary(UnAlu, Box<Expr>),
    Binary(BinAlu, Box<(Expr, Expr)>),
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum FBinOp {
    And,
    Or,
    Implies,
    Iff,
    AndAsym,
}

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum QType {
    Exists,
    Forall,
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Formula {
    Val(bool),
    Not(Box<Formula>),
    Bin(FBinOp, Box<(Formula, Formula)>),
    Quant(QType, Ident, Box<Formula>),
    Rel(Cc, Expr, Expr),
    IsBuffer(Ident, Expr),
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Logic {
    Assert(Formula),
```

```
    Require(Formula),
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Line {
    Label(Label),
    Logic(Logic),
    Stmt(Stmt),
    Cont(Cont),
}

pub struct Module {
    pub requires: Vec<Formula>,
    pub ensures: Vec<Formula>,
    pub lines: Vec<Line>,
}
```

A.4 src/formula.rs

```
//! A stateful builder for formulas.

use std::collections::HashMap;

use crate::ast::*;

#[derive(Default)]
pub struct FormulaBuilder {
    id_counters: HashMap<String, usize>,
}

impl FormulaBuilder {
    pub fn new() -> Self {
        Self::default()
    }

    pub fn top(&self) -> Formula {
        Formula::Val(true)
    }

    pub fn bot(&self) -> Formula {
        Formula::Val(false)
    }

    pub fn not(&self, f: Formula) -> Formula {
        Formula::Not(Box::new(f))
    }

    pub fn and(&self, a: Formula, b: Formula) -> Formula {
        Formula::Bin(FBinOp::And, Box::new((a, b)))
    }

```

```
33      pub fn asym_and(&self, a: Formula, b: Formula) -> Formula {
34          Formula::Bin(FBinOp::AndAsym, Box::new((a, b)))
35      }
36
37      pub fn or(&self, a: Formula, b: Formula) -> Formula {
38          Formula::Bin(FBinOp::Or, Box::new((a, b)))
39      }
40
41      pub fn implies(&self, a: Formula, b: Formula) -> Formula {
42          Formula::Bin(FBinOp::Implies, Box::new((a, b)))
43      }
44
45      pub fn iff(&self, a: Formula, b: Formula) -> Formula {
46          Formula::Bin(FBinOp::Iff, Box::new((a, b)))
47      }
48
49      pub fn forall(&self, ident: Ident, f: Formula) -> Formula {
50          Formula::Quant(QType::Forall, ident, Box::new(f))
51      }
52
53      pub fn exists(&self, ident: Ident, f: Formula) -> Formula {
54          Formula::Quant(QType::Exists, ident, Box::new(f))
55      }
56
57      pub fn replace(&self, prev: &Ident, new: &Ident, f: &Formula) -> Option<Formula> {
58          match f {
59              Formula::Not(inner) => {
60                  let res = self.replace(prev, new, inner)?;
61                  Some(Formula::Not(Box::new(res)))
62              }
63              Formula::Bin(op, fs) => {
64                  let a = self.replace(prev, new, &fs.0);
65                  let b = self.replace(prev, new, &fs.1);
66                  if a.is_some() || b.is_some() {
67                      Some(Formula::Bin(
68                          *op,
69                          Box::new((a.unwrap_or(fs.0.clone()), b.unwrap_or(fs.1.clone()))),
70                      ))
71                  } else {
72                      None
73                  }
74              }
75              Formula::Quant(qtype, qvar, inner) => {
76                  if prev != qvar {
77                      let res = self.replace(prev, new, inner)?;
78                      Some(Formula::Quant(*qtype, qvar.clone(), Box::new(res)))
79                  } else {
80                      None
81                  }
82              }
83              Formula::Rel(r, e1, e2) => {
84                  let a = self.replace_expr(prev, new, e1);
85                  let b = self.replace_expr(prev, new, e2);
86                  if a.is_some() || b.is_some() {
87                      Some(Formula::Rel(
```

```
                        *r,
                        a.unwrap_or(e1.clone()),
                        b.unwrap_or(e2.clone()),
                    ))
                } else {
                    None
                }
            }
            Formula::IsBuffer(ptr, sz) => {
                let new_ptr = if ptr == prev { Some(new.clone()) } else { None };
                let new_sz = self.replace_expr(prev, new, sz);
                if new_ptr.is_some() || new_sz.is_some() {
                    Some(Formula::IsBuffer(
                        new_ptr.unwrap_or(ptr.clone()),
                        new_sz.unwrap_or(sz.clone()),
                    ))
                } else {
                    None
                }
            }
            Formula::Val(_) => None,
        }
    }

    pub fn replace_expr(&self, prev: &Ident, new: &Ident, e: &Expr) -> Option<Expr> {
        match e {
            Expr::Var(x) => {
                if x == prev {
                    Some(Expr::Var(new.clone()))
                } else {
                    None
                }
            }
            Expr::Unary(op, inner) => {
                let res = self.replace_expr(prev, new, inner)?;
                Some(Expr::Unary(*op, Box::new(res)))
            }
            Expr::Binary(op, es) => {
                let a = self.replace_expr(prev, new, &es.0);
                let b = self.replace_expr(prev, new, &es.1);
                if a.is_some() || b.is_some() {
                    Some(Expr::Binary(
                        *op,
                        Box::new((a.unwrap_or(es.0.clone()), b.unwrap_or(es.1.clone()))),
                    ))
                } else {
                    None
                }
            }
            Expr::Val(_) => None,
        }
    }

    pub fn rel(&self, cc: Cc, a: Expr, b: Expr) -> Formula {
        Formula::Rel(cc, a, b)
```

```
    }

    pub fn eq(&self, a: Expr, b: Expr) -> Formula {
        Formula::Rel(Cc::Eq, a, b)
    }

    /// Generate a new, unique variable.
    pub fn var(&mut self, mut ident: Ident) -> (Expr, Ident) {
        let counter = self.id_counters.entry(ident.clone()).or_insert(0);
        ident.extend(format!("{counter}").chars());
        *counter += 1;
        (Expr::Var(ident.clone()), ident)
    }

    /// Generate a non-unique expression representing [ident].
    pub fn var_ident(&self, ident: Ident) -> Expr {
        Expr::Var(ident)
    }

    /// Get the variable representing a register.
    pub fn reg(&self, reg: Reg) -> (Expr, Ident) {
        let id = format!("r{}", reg.get());
        (Expr::Var(id.clone()), id)
    }

    pub fn val(&self, i: Imm) -> Expr {
        Expr::Val(i)
    }

    pub fn unop(&self, op: UnAlu, e: Expr) -> Expr {
        Expr::Unary(op, Box::new(e))
    }

    pub fn binop(&self, op: BinAlu, a: Expr, b: Expr) -> Expr {
        Expr::Binary(op, Box::new((a, b)))
    }

    pub fn is_buffer(&self, ptr: Ident, size: Expr) -> Formula {
        Formula::IsBuffer(ptr, size)
    }
}
```

A.5 src/cfg.rs

```
//! A processed AST, ready for VC-generation.
//! It currently only supports 64-bit operations.

use std::{
    collections::HashMap,
    fmt::{self, Display, Formatter},
    mem::swap,
```

```rust
};

pub use crate::ast::{
    BinAlu, Cc, Cont, Expr, Formula, Ident, Imm, Label, MemRef, Offset, Reg, RegImm, Stmt, UnAlu,
    WordSize,
};

use crate::{
    ast::{Line, Logic, Module},
    formula::FormulaBuilder,
};

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Continuation {
    Exit,
    Jmp(Label),
    Jcc(Cc, Reg, RegImm, Label, Label),
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Block {
    pub require: Option<Formula>,
    pub body: Vec<Stmt>,
    pub next: Continuation,
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Cfg {
    pub requires: Formula,
    pub ensures: Formula,
    pub start: Label,
    pub blocks: HashMap<Label, Block>,
}

pub enum ConvertErr {
    NoExit,
    JumpBounds { target: usize, bound: usize },
    NoLabel(String),
    Unsupported(Stmt),
    MisplacedRequire,
    DuplicateLabel(String),
}

impl Display for ConvertErr {
    fn fmt(&self, f: &mut Formatter<'_>) -> fmt::Result {
        match self {
            ConvertErr::NoExit => f.write_fmt(format_args!("Last instruction must be \"exit\"")),
            ConvertErr::NoLabel(label) => {
                f.write_fmt(format_args!("Jump target \"{label}\" doesn't exist"))
            }
            ConvertErr::JumpBounds { target, bound } => {
                f.write_fmt(format_args!("Jump target {target} outside bound {bound}"))
            }
            ConvertErr::Unsupported(instr) => {
                f.write_fmt(format_args!("Unsupported instruction: {instr:?}"))
```

```
            }
            ConvertErr::MisplacedRequire => {
                f.write_str("Requirements can only be placed at the start of blocks")
            }
            ConvertErr::DuplicateLabel(label) => {
                f.write_fmt(format_args!("Duplicate label \"{label}\""))
            }
        }
    }
}

struct State {
    blocks: HashMap<Label, Block>,
    label_aliases: HashMap<String, String>,
    label_counter: usize,
    label: String,
    require: Option<Formula>,
    body: Vec<Stmt>,
}

impl State {
    fn new() -> Self {
        Self {
            blocks: HashMap::new(),
            label_aliases: HashMap::new(),
            label: "@0".to_owned(),
            label_counter: 0,
            require: None,
            body: Vec::new(),
        }
    }

    fn finish(&mut self, next: Continuation) -> Result<(), ConvertErr> {
        if self.blocks.contains_key(&self.label) {
            return Err(ConvertErr::DuplicateLabel(self.label.clone()));
        }
        let mut label = "".to_owned();
        let mut require = None;
        let mut body = Vec::new();
        swap(&mut self.label, &mut label);
        swap(&mut self.require, &mut require);
        swap(&mut self.body, &mut body);
        self.blocks.insert(
            label,
            Block {
                require,
                body,
                next,
            },
        );
        Ok(())
    }

    fn change_label(&mut self, l: Label) {
        let mut tmp = l.clone();
```

```
        swap(&mut self.label, &mut tmp);
        self.label_aliases.insert(tmp, l);
    }

    fn next_label(&mut self) -> Label {
        self.label_counter += 1;
        format!("@{}", self.label_counter)
    }

    fn resolve_aliases(&mut self) {
        let resolve = |target: &mut Label| {
            if let Some(l) = self.label_aliases.get(target) {
                *target = l.clone();
            }
        };
        for block in self.blocks.values_mut() {
            match &mut block.next {
                Continuation::Jcc(_, _, _, target_t, target_f) => {
                    resolve(target_t);
                    resolve(target_f);
                }
                Continuation::Jmp(target) => {
                    resolve(target);
                }
                _ => (),
            }
        }
    }
}

impl Cfg {
    pub fn create(ast: Module, f: &mut FormulaBuilder) -> Result<Cfg, ConvertErr> {
        let mut state = State::new();
        if ast.lines.last() != Some(&Line::Cont(Cont::Exit)) {
            return Err(ConvertErr::NoExit);
        }
        for line in ast.lines {
            match line {
                Line::Label(l) => {
                    if !state.body.is_empty() {
                        state.finish(Continuation::Jmp(l.clone()))?;
                    }
                    state.change_label(l);
                }
                Line::Logic(Logic::Assert(a)) => state.body.push(Stmt::Assert(a)),
                Line::Logic(Logic::Require(i)) => {
                    if state.body.is_empty() {
                        state.require = match state.require {
                            // TODO: Normal or asymmetric conjugation?
                            Some(pa) => Some(f.asym_and(pa, i)),
                            None => Some(i),
                        };
                    } else {
                        return Err(ConvertErr::MisplacedRequire);
                    }
```

```
                }
                Line::Stmt(i) => state.body.push(i),
                Line::Cont(c) => match c {
                    // End of blocks
                    Cont::Jmp(t) => {
                        state.finish(Continuation::Jmp(t))?;
                        let next_label = state.next_label();
                        state.change_label(next_label)
                    }
                    Cont::Jcc(cc, reg, reg_imm, target) => {
                        let next_label = state.next_label();
                        state.finish(Continuation::Jcc(
                            cc,
                            reg,
                            reg_imm,
                            target,
                            next_label.clone(),
                        ))?;
                        state.change_label(next_label);
                    }
                    Cont::Exit => state.finish(Continuation::Exit)?,
                },
            }
        }
        state.resolve_aliases();

        let requires = ast.requires.into_iter().fold(f.top(), |a, b| f.and(a, b));
        let ensures = ast.ensures.into_iter().fold(f.top(), |a, b| f.and(a, b));
        Ok(Cfg {
            requires,
            ensures,
            start: state
                .label_aliases
                .get("@0")
                .unwrap_or(&"@0".to_owned())
                .to_owned(),
            blocks: state.blocks,
        })
    }
}
```

A.6 src/parse.rs

```
//! Parsing of eBPF assembly.

use nom::{
    branch::alt, bytes::complete::tag, character::complete::*, combinator::*, multi::*,
    sequence::*, IResult, Parser,
};

use crate::ast::*;
```

```rust

#[cfg(test)]
#[rustfmt::skip]
mod tests;

// TODO: Improve the whitespace story.

type Res<'a, O> = IResult<&'a str, O>;

// Tokens

fn num(i: &str) -> Res<i64> {
    let num_dec = map_res(
        recognize(many1(terminated(one_of("0123456789"), many0(char('_'))))),
        |out: &str| str::replace(out, "_", "").parse::<i64>(),
    );

    let num_bin = map_res(
        preceded(
            tag("0b"),
            recognize(many1(terminated(one_of("01"), many0(char('_'))))),
        ),
        |out: &str| i64::from_str_radix(&str::replace(out, "_", ""), 2),
    );

    let num_hex = map_res(
        preceded(
            tag("0x"),
            recognize(many1(terminated(
                one_of("0123456789abcdefABCDEF"),
                many0(char('_')),
            ))),
        ),
        |out: &str| i64::from_str_radix(&str::replace(out, "_", ""), 16),
    );

    alt((num_hex, num_bin, num_dec))(i)
}

fn ident(i: &str) -> Res<&str> {
    recognize(tuple((
        opt(tag("_")),
        alpha1,
        many0_count(alt((alphanumeric1, tag("_")))),
    )))(i)
}

// Instruction parsing

fn reg(i: &str) -> Res<Reg> {
    map_opt(
        preceded(
            char('r'),
            alt((
                tag("10"),
```

```
                tag("0"),
                tag("1"),
                tag("2"),
                tag("3"),
                tag("4"),
                tag("5"),
                tag("6"),
                tag("7"),
                tag("8"),
                tag("9"),
            )),
        ),
        |num: &str| Some(Reg::new(num.parse::<u8>().ok()?))?,
    )(i)
}

fn imm(i: &str) -> Res<Imm> {
    pair(opt(terminated(alt((char('+'), char('-'))), space0)), num)
        .map(|(sign, n)| match sign {
            Some('+') | None => n,
            Some('-') => -n,
            _ => unreachable!(),
        })
        .parse(i)
}

fn reg_imm(i: &str) -> Res<RegImm> {
    alt((map(reg, RegImm::Reg), map(imm, RegImm::Imm)))(i)
}

fn offset(i: &str) -> Res<Offset> {
    alt((
        preceded(pair(char('+'), space0), imm),
        preceded(pair(char('-'), space0), map(num, |n| -n)),
    ))(i)
}

fn alu_size(i: &str) -> Res<WordSize> {
    alt((
        value(WordSize::B32, tag("32")),
        value(WordSize::B64, opt(tag("64"))),
    ))(i)
}

/// Separator between components of an instruction
fn isep(i: &str) -> Res<()> {
    value(
        (),
        verify(
            recognize(tuple((space0, opt(char(',')), space0))),
            |res: &str| !res.is_empty(),
        ),
    )(i)
}

```

```
macro_rules! instr {
    ( $head:expr, $first:expr $(, $($tail:expr),* )? ) => {
        tuple((terminated($head, space1), $first $(, $(preceded(isep, $tail)),* )? ))
    };
}

fn un_alu(i: &str) -> Res<UnAlu> {
    alt((
        value(UnAlu::Neg, tag("neg")),
        value(UnAlu::Le, tag("le")),
        value(UnAlu::Be, tag("be")),
    ))(i)
}

fn bin_alu(i: &str) -> Res<BinAlu> {
    alt((
        value(BinAlu::Mov, tag("mov")),
        value(BinAlu::Add, tag("add")),
        value(BinAlu::Sub, tag("sub")),
        value(BinAlu::Mul, tag("mul")),
        value(BinAlu::Div, tag("div")),
        value(BinAlu::Mod, tag("mod")),
        value(BinAlu::And, tag("and")),
        value(BinAlu::Or, tag("or")),
        value(BinAlu::Xor, tag("xor")),
        value(BinAlu::Lsh, tag("lsh")),
        value(BinAlu::Rsh, tag("rsh")),
        value(BinAlu::Arsh, tag("arsh")),
    ))(i)
}

fn unary(i: &str) -> Res<Stmt> {
    instr!(pair(un_alu, alu_size), reg)
        .map(|((op, size), reg)| Stmt::Unary(size, op, reg))
        .parse(i)
}

fn binary(i: &str) -> Res<Stmt> {
    instr!(pair(bin_alu, alu_size), reg, reg_imm)
        .map(|((op, size), reg, reg_imm)| Stmt::Binary(size, op, reg, reg_imm))
        .parse(i)
}

fn mem_size(i: &str) -> Res<WordSize> {
    alt((
        value(WordSize::B8, char('b')),
        value(WordSize::B16, char('h')),
        value(WordSize::B32, char('w')),
        value(WordSize::B64, tag("dw")),
    ))(i)
}

fn mem_ref(i: &str) -> Res<MemRef> {
    let inner = map(
        tuple((reg, space0, opt(offset), space0)),
```

```
        |(reg, _, offset, _)| MemRef(reg, offset.unwrap_or(0)),
    );
    delimited(terminated(char('['), space0), inner, char(']'))(i)
}

fn load(i: &str) -> Res<Stmt> {
    map(
        instr!(preceded(tag("ldx"), mem_size), reg, mem_ref),
        |(size, reg, mem_ref)| Stmt::Load(size, reg, mem_ref),
    )(i)
}

fn store(i: &str) -> Res<Stmt> {
    instr!(
        preceded(alt((tag("stx"), tag("st"))), mem_size),
        mem_ref,
        reg_imm
    )
    .map(|(size, mref, reg_imm)| Stmt::Store(size, mref, reg_imm))
    .parse(i)
}

fn cont(i: &str) -> Res<Cont> {
    let cc = alt((
        value(Cc::Eq, tag("eq")),
        value(Cc::Gt, tag("gt")),
        value(Cc::Ge, tag("ge")),
        value(Cc::Lt, tag("lt")),
        value(Cc::Le, tag("le")),
        value(Cc::Set, tag("set")),
        value(Cc::Ne, tag("ne")),
        value(Cc::Sgt, tag("sgt")),
        value(Cc::Sge, tag("sge")),
        value(Cc::Slt, tag("slt")),
        value(Cc::Sle, tag("sle")),
    ));
    let jcc = map(
        instr!(
            preceded(char('j'), cc),
            reg,
            reg_imm,
            ident.map(|id| id.to_owned())
        ),
        |(cc, lhs, rhs, target)| Cont::Jcc(cc, lhs, rhs, target),
    );

    let jmp = map(
        preceded(
            pair(alt((tag("ja"), tag("jmp"))), space1),
            ident.map(|id| id.to_owned()),
        ),
        Cont::Jmp,
    );
    let exit = value(Cont::Exit, tag("exit"));

```

```
    alt((exit, jmp, jcc))(i)
}

fn stmt(i: &str) -> Res<Stmt> {
    let call = map(preceded(pair(tag("call"), space1), imm), Stmt::Call);
    let load_imm = map(instr!(tag("lddw"), reg, imm), |(_, reg, imm)| {
        Stmt::LoadImm(reg, imm)
    });
    // Missing: LoadMapFd
    alt((unary, binary, load, load_imm, store, call))(i)
}

// Assertion parsing
fn parens<'a, T>(p: impl FnMut(&'a str) -> Res<'a, T>) -> impl FnMut(&'a str) -> Res<'a, T> {
    delimited(
        terminated(char('('), space0),
        terminated(p, space0),
        terminated(char(')'), space0),
    )
}

fn expr(i: &str) -> Res<Expr> {
    let unary = tuple((un_alu, parens(expr))).map(|(op, inner)| Expr::Unary(op, Box::new(inner)));
    let binary = tuple((
        terminated(bin_alu, space0),
        parens(tuple((expr, space0, char(','), space0, expr))),
    ))
    .map(|(op, (a, _, _, _, b))| Expr::Binary(op, Box::new((a, b))));

    alt((
        binary,
        unary,
        imm.map(Expr::Val),
        ident.map(|id| Expr::Var(id.to_owned())),
    ))(i)
}

fn formula(i: &str) -> Res<Formula> {
    let parenthesized = parens(formula);
    let val = alt((
        value(Formula::Val(true), tag("true")),
        value(Formula::Val(false), tag("false")),
    ));
    let not = preceded(tag("not"), parens(formula)).map(|inner| Formula::Not(Box::new(inner)));

    let bin_op = alt((
        value(FBinOp::And, tag("/\\")),
        value(FBinOp::Or, tag("\\/")),
        value(FBinOp::Implies, tag("->")),
        value(FBinOp::Iff, tag("<->")),
        value(FBinOp::AndAsym, tag("&&")),
    ));

    let binary = tuple((
        bin_op,
```

```
        parens(tuple((formula, space0, char(','), space0, formula))),
    ))
    .map(|(op, (a, _, _, _, b))| Formula::Bin(op, Box::new((a, b))));

    let quantifier = alt((
        value(QType::Forall, tag("forall")),
        value(QType::Exists, tag("exists")),
    ));

    let quant = tuple((quantifier, space1, ident, char('.'), space0, formula)).map(
        |(quantifier, _, name, _, _, inner)| {
            Formula::Quant(quantifier, name.to_owned(), Box::new(inner))
        },
    );

    let rel_op = alt((
        value(Cc::Eq, tag("=")),
        value(Cc::Ne, tag("<>")),
        value(Cc::Ge, tag(">=")),
        value(Cc::Le, tag("<=")),
        value(Cc::Lt, tag("<")),
        value(Cc::Gt, tag(">")),
        // TODO: signed comparisons and 'set'
    ));

    let rel = tuple((expr, space0, rel_op, space0, expr))
        .map(|(e1, _, op, _, e2)| Formula::Rel(op, e1, e2));

    let is_buffer = tuple((
        tag("is_buffer"),
        space0,
        parens(tuple((ident, char(','), space0, expr, space0))),
    ))
    .map(|(_, _, (id, _, _, e, _))| Formula::IsBuffer(id.to_owned(), e));
    alt((parenthesized, val, not, binary, quant, rel, is_buffer))(i)
}

fn formula_line(i: &str) -> Res<Logic> {
    preceded(
        pair(tag(";#"), space0),
        alt((
            preceded(pair(tag("assert"), space0), map(formula, Logic::Assert)),
            preceded(pair(tag("req"), space0), map(formula, Logic::Require)),
        )),
    )(i)
}

// Structural parsing

fn line_sep(i: &str) -> Res<()> {
    value(
        (),
        many1(tuple((
            space0,
            opt(pair(
```

```
                terminated(char(';'), peek(satisfy(|c| c != '#'))),
                many0(satisfy(|c| c != '\n')),
            )),
            newline,
            space0,
        ))),
    )(i)
}

fn label(i: &str) -> Res<Label> {
    terminated(ident, pair(space0, tag(":")))
        .map(|l| l.to_owned())
        .parse(i)
}

fn line(i: &str) -> Res<Line> {
    alt((
        label.map(Line::Label),
        formula_line.map(Line::Logic),
        stmt.map(Line::Stmt),
        cont.map(Line::Cont),
    ))(i)
}

pub fn module(i: &str) -> Res<Module> {
    let requirement = preceded(tuple((tag(";#"), space0, tag("requires"), space0)), formula);
    let ensurance = preceded(tuple((tag(";#"), space0, tag("ensures"), space0)), formula);
    let components = tuple((
        many0(terminated(requirement, line_sep)),
        many0(terminated(ensurance, line_sep)),
        preceded(space0, separated_list0(line_sep, line)),
    ));
    delimited(
        opt(line_sep),
        map(components, |(rs, es, ls)| Module {
            lines: ls,
            requires: rs,
            ensures: es,
        }),
        pair(opt(line_sep), eof),
    )(i)
}
```

A.7 src/whyml.rs

```
//! WhyML generation from formulas.

use crate::ast::*;

pub struct Conditions(pub Vec<(String, Formula)>);

```

```rust
impl std::fmt::Display for Conditions {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        f.write_str(
            "use mach.int.UInt64\n\
             use int.Int\n\
             use int.ComputerDivision\n\
             predicate is_buffer (p: uint64) (s: uint64)\n\n",
        )?;
        for (name, goal) in self.0.iter() {
            f.write_fmt(format_args!(
                "goal {name}: forall r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 : uint64 . {goal}\n\n"
            ))?;
        }
        Ok(())
    }
}

impl std::fmt::Display for Expr {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Expr::Val(imm) => f.write_fmt(format_args!("{imm}")),
            Expr::Var(ident) => f.write_str(ident),
            Expr::Unary(op, e) => {
                let op_str = match op {
                    UnAlu::Neg => "neg",
                    UnAlu::Le => "le",
                    UnAlu::Be => "be",
                };
                f.write_fmt(format_args!("{op_str}({e})"))
            }
            Expr::Binary(op, es) => {
                let (e1, e2) = &**es;
                let op_str = match op {
                    BinAlu::Mov => return f.write_fmt(format_args!("{e2}")),
                    BinAlu::Add => "+",
                    BinAlu::Sub => "-",
                    BinAlu::Mul => "*",
                    BinAlu::Div => return f.write_fmt(format_args!("div {e1} {e2}")),
                    BinAlu::Mod => return f.write_fmt(format_args!("mod {e1} {e2}")),
                    BinAlu::And => "&",
                    BinAlu::Or => "|",
                    BinAlu::Xor => "^",
                    BinAlu::Lsh => "<<",
                    BinAlu::Rsh => ">>",
                    BinAlu::Arsh => ">>>",
                };
                f.write_fmt(format_args!("({e1} {op_str} {e2})"))
            }
        }
    }
}

impl std::fmt::Display for QType {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        f.write_str(match self {
```

```
            QType::Exists => "exists",
            QType::Forall => "forall",
        })
    }
}

impl std::fmt::Display for Formula {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Formula::Val(b) => f.write_str(if *b { "true" } else { "false" }),
            Formula::Not(form) => f.write_fmt(format_args!("not ({form})")),
            Formula::Bin(op, fs) => {
                let (f1, f2) = &**fs;
                let op_str = match op {
                    FBinOp::And => "/\\",
                    FBinOp::Or => "\\/",
                    FBinOp::Implies => "->",
                    FBinOp::Iff => "<->",
                    FBinOp::AndAsym => "&&",
                };
                f.write_fmt(format_args!("({f1} {op_str} {f2})"))
            }
            Formula::Quant(q, id, form) => {
                f.write_fmt(format_args!("({q} {id} : uint64 . {form})"))
            }
            Formula::Rel(rel, e1, e2) => {
                let rel_str = match rel {
                    Cc::Eq => "=",
                    Cc::Gt => ">",
                    Cc::Ge => ">=",
                    Cc::Lt => "<",
                    Cc::Le => "<=",
                    Cc::Set => todo!(),
                    Cc::Ne => "<>",
                    Cc::Sgt => todo!(),
                    Cc::Sge => todo!(),
                    Cc::Slt => todo!(),
                    Cc::Sle => todo!(),
                };
                f.write_fmt(format_args!("({e1} {rel_str} {e2})"))
            }
            Formula::IsBuffer(ptr, sz) => f.write_fmt(format_args!("is_buffer {ptr} {sz}")),
        }
    }
}
```

B verifier.c

```
#define _GNU_SOURCE
#include <err.h>
#include <stdint.h>
```

```c
#include <stdio.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <stdlib.h>

#include <bpf/bpf.h>
#include "bpf_insn.h"


// Wrapper for BPF syscall
static long bpf_(int cmd, union bpf_attr *attrs) {
  return syscall(__NR_bpf, cmd, attrs, sizeof(*attrs));
}

// ----- Shared memory interface -----
// Maps are a generic interface
static uint32_t map_create(uint64_t value_size) {
  union bpf_attr create_map_attrs = {
    .map_type    = BPF_MAP_TYPE_ARRAY, // map implementation
    .key_size    = 4,
    .value_size  = value_size,
    .max_entries = 16
  };

  uint32_t map_fd = bpf_(BPF_MAP_CREATE, &create_map_attrs);
  if (map_fd == -1)
    err(1, "map create");

  return map_fd;
}

static void array_set(int mapfd, uint32_t key, void *value) {
  union bpf_attr attr = {
    .map_fd = mapfd,
    .key    = (uint64_t)&key,
    .value  = (uint64_t)value,
    .flags  = BPF_ANY,
  };

  long res = bpf_(BPF_MAP_UPDATE_ELEM, &attr);
  if (res)
    err(1, "map update elem");
}

static uint32_t array_get(int map_fd, uint32_t key) {
  uint64_t ret_val;
  union bpf_attr lookup_map = {
    .map_fd = map_fd,
    .key    = (uint64_t)&key,
    .value  = (uint64_t)&ret_val
  };

  int res = bpf_(BPF_MAP_LOOKUP_ELEM, &lookup_map);
```

```
  if (res)
    err(1, "map lookup elem");
  return ret_val;
}

// Simple sized buffer.
typedef struct Buffer {
    char* data;
    long size;
} Buffer;

Buffer read_file(char* path) {
    char *source = NULL;
    long bufsize = 0;
    FILE *fp = fopen(path, "r");
    if (fp != NULL) {
        /* Go to the end of the file. */
        if (fseek(fp, 0L, SEEK_END) == 0) {
            /* Get the size of the file. */
            bufsize = ftell(fp);
            if (bufsize <= 0) { err(1, "Error reading file"); }

            /* Allocate our buffer to that size. */
            source = malloc(sizeof(char) * (bufsize + 1));

            /* Go back to the start of the file. */
            if (fseek(fp, 0L, SEEK_SET) != 0) { err(1, "Error reading file"); }

            /* Read the entire file into memory. */
            size_t newLen = fread(source, sizeof(char), bufsize, fp);
            if ( ferror( fp ) != 0 ) {
                err(1, "Error reading file");
            } else {
                source[newLen++] = '\0'; /* Just to be safe. */
            }
        } else {
            err(1, "Error reading file");
        }
        fclose(fp);
    } else {
        err(1, "failed to open file");
    }
    Buffer res = {
        .data = source,
        .size = bufsize
    };
    return res;
}

Buffer concat(Buffer a, Buffer b) {
    long new_size = a.size + b.size;
    char* new_data = malloc(new_size);
    memcpy(new_data, a.data, a.size);
    memcpy(new_data + a.size, b.data, b.size);
```

```
    Buffer res = {
        .data = new_data,
        .size = new_size
    };
    return res;
}

// ----- Main code -----
int main(int argc, char** argv) {

  if (argc != 2 && argc != 3) {
      fprintf(stderr, "test if an eBPF module is accepted by the verifier\nusage: %s [EBPF BINARY]
      ↪  [--log]\n", argv[0]);
      return 1;
  }

  uint32_t ctx_size = 64;
  printf("Size of input buffer: %d\n", ctx_size);
  uint32_t ctx_map_fd = map_create(ctx_size);
  uint32_t size_map_fd = map_create(8);

  if (size_map_fd == -1 || ctx_map_fd == -1) {
      err(1, "Failed to create maps\n");
  } else {
      printf("Created maps\n");
  }

  // I don't think that this prelude is entirely correct.
  // It should be possible to get the value size through some weird undocumented instruction.
  struct bpf_insn header[] = {
      // Load buffer, put pointer in r9.
      BPF_LD_MAP_FD(BPF_REG_1, ctx_map_fd),
      BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
      BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),
      BPF_ST_MEM(BPF_W, BPF_REG_2, 0, 0),
      BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, 1),
      BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, +2),
      BPF_MOV64_IMM(BPF_REG_0, 1),
      BPF_EXIT_INSN(),
      BPF_MOV64_REG(BPF_REG_9, BPF_REG_0),
      BPF_MOV64_IMM(BPF_REG_0, 0),

      // Load first integer into r6.
      BPF_LD_MAP_FD(BPF_REG_1, size_map_fd),
      BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
      BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),
      BPF_ST_MEM(BPF_W, BPF_REG_2, 0, 0), // index = 0
      BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, 1),
      BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, +2),
      BPF_MOV64_IMM(BPF_REG_0, 1),
      BPF_EXIT_INSN(),
      BPF_LDX_MEM(BPF_DW, BPF_REG_6, BPF_REG_0, 0),

      // Load second integer into r7.
```

```
    BPF_LD_MAP_FD(BPF_REG_1, size_map_fd),
    BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
    BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),
    BPF_ST_MEM(BPF_W, BPF_REG_2, 0, 1), // index = 1
    BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, 1),
    BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, +2),
    BPF_MOV64_IMM(BPF_REG_0, 1),
    BPF_EXIT_INSN(),
    BPF_LDX_MEM(BPF_DW, BPF_REG_7, BPF_REG_0, 0),

    // Load third integer into r8.
    BPF_LD_MAP_FD(BPF_REG_1, size_map_fd),
    BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
    BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),
    BPF_ST_MEM(BPF_W, BPF_REG_2, 0, 1), // index = 2
    BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, 1),
    BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, +2),
    BPF_MOV64_IMM(BPF_REG_0, 1),
    BPF_EXIT_INSN(),
    BPF_LDX_MEM(BPF_DW, BPF_REG_8, BPF_REG_0, 0),

    // Put buffer pointer in r1, size in r2.
    BPF_MOV64_REG(BPF_REG_1, BPF_REG_9),
    BPF_MOV64_REG(BPF_REG_2, BPF_REG_6),
    BPF_MOV64_REG(BPF_REG_3, BPF_REG_7),
    BPF_MOV64_REG(BPF_REG_4, BPF_REG_8),
};

Buffer header_buf = {
    .data = (char*)header,
    .size = sizeof(header),
};
Buffer prog_buf = read_file(argv[1]);
printf("Loaded program buffer (%ld bytes)\n", prog_buf.size);

Buffer full_buf = concat(header_buf, prog_buf);

// load the program
char verifier_log[100000];
union bpf_attr create_prog_attrs = {
  .prog_type = BPF_PROG_TYPE_SOCKET_FILTER,
  .insn_cnt = full_buf.size / 8,
  .insns = (uint64_t)full_buf.data,
  .license = (uint64_t) "GPL",
  .log_level = 2,
  .log_size = sizeof(verifier_log),
  .log_buf = (uint64_t)verifier_log
};
int progfd = bpf_(BPF_PROG_LOAD, &create_prog_attrs);

if (argc == 3 && strcmp(argv[2], "--log") == 0) {
  puts(verifier_log);
}

```

```
  // If verification doesn't accept the program, this is where we get the error
  if (progfd == -1) {
    perror("Program denied\n\n");
    return 1;
  }
  printf("Program accepted\n");
  return 0;
}
```