eJitk — Extending Jitk to eBPF

Louis Sobel — 6.888 Project

May 21, 2015

1 Introduction

This paper gives the background and description of my 6.888 project to expand Jitk [1] to support eBPF. The BSD Packet Filter (BPF) language [2] was first introduced into the Linux kernel as a way to efficiently run user-provided code to filter network packets within the kernel. Since then, kernel developers have built a new version, *extended BPF* (eBPF), with additional capabilities and uses [3].

Jitk is a project that provides a verified compiler for BPF programs. The project I describe in this paper builds on Jitk to provide a verified compiler for the new version of BPF: eBPF. There are a number of significant differences between BPF and eBPF. I account for some of these differences in the specification, implementation, and proofs of the compiler.

The rest of the paper is as follows. Section 2 gives further background on BPF, Jitk, and eBPF. Section 3 discusses the work for the eJitk project. Section 4 lists remaining work. Section 5 reflects on my experience with Coq, and Section 6 concludes.

2 Background

This section provides background information on BPF and its use in the kernel, Jitk and its implementation approach, and eBPF and its differences with BPF.

2.1 BPF

BPF was introduced as a way to efficiently run filters passed from userspace within the kernel network stack. It is a bytecode based language whose interpreter presents a simple virtual machine with two registers, a small scratch memory region, and the input packet itself. There are ALU operations, loads and stores to access memory, and relative jumps whose offsets are limited to non-negative numbers – this ensures termination.

Since being introduced into the Linux kernel, BPF's role has expanded beyond interpreted packet filtering. The kernel contains just-in-time (JIT) compilers for BPF[4], which increase the performance of BPF programs by compiling them to a native instruction set. Additionally, BPF is used in the Seccomp subsystem to provide applications fine-grained control over which system calls they are permitted to make.

2.2 Jitk

Jitk targets the Seccomp use case of BPF. It includes a verified JIT compiler that takes BPF as input and generates native assembly code which is proven to have the same semantics.

Jitk consists of number of components written in Coq. Rather than verifying the BPF compiler all the way to assembly, Jitk leverages the verified C compiler CompCert[5]. CompCert includes an intermediate language Cminor. Jitk translates BPF to Cminor, relying on CompCert to translate Cminor to native code.

To do this, Jitk defines three things: a formal specification of the BPF language, a filter for statically checking BPF programs, and a function that performs the actual eBPF $\rightarrow$ Cminor translation. Jitk also has two proofs of the correctness of this translation. The first is that the compiler preserves the semantics of BPF. This proof shows that the generated Cminor code preserves the small-step semantics of BPF as defined in the specification.

The second proof shows that if the filter accepts a BPF program, then that program terminates. This proof requires that no undefined behavior passes the filter. For example, the filter rejects jumps with negative offsets and out-of-bound memory accesses, ensuring defined behavior. This proof does not detect a filter that is too strict.

2.3 eBPF

This section describes the semantic differences between BPF and eBPF as well as how eBPF is used in the Linux kernel.

2.3.1 Comparison with BPF

eBPF is fundamentally similar to BPF, but has a number of significant differences. Like BPF, eBPF presents a virtual machine with a simple bytecode interface. There are registers, ALU operations, loads, stores, and jumps. The specification for eBPF is available in the Linux source tree[6]. eBPF is different from BPF in many ways. Four of the main differences involve registers, word size, kernel function calls, and jump instructions.

Registers BPF has only two registers: `reg_a`, an accumulator, and `reg_x`, a memory index register. eBPF has eleven, `R[0-10]`. `R10` is a read-only frame pointer which represents the start of the code's available scratch memory. These registers are designed to each map directly to a native machine register for simple compilation.

Word Size eBPF's virtual machine is 64-bits, whereas BPF's is 32-bits. This is for easy compilation to 64-bit architectures like ARMv8 and x86_64. 32-bit operation is preserved for compatibility – each of the eleven 64-bit registers has an implicit 32-bit sub-register. There are two corresponding disjoint classes of ALU operations.

Kernel Function Calls Unlike BPF programs, eBPF programs are capable of calling into the kernel using a set of predefined functions. The purpose of this is to make the language extensible and flexible. The calling convention was chosen to make kernel function calls zero-overhead for compiled programs.

Jump Instructions Jump instructions in BPF always do unsigned comparisons and have two offsets: one for if the result of the comparison is true and the other for if the result is false. eBPF adds jumps with signed comparisons and changes all jumps to have only one offset – used if the result of the comparison is true – and otherwise fall through to the next instruction.

2.3.2 Use in Linux

The use of eBPF in Linux has evolved since eBPF was introduced. As described in [7], eBPF was first integrated into the kernel as an internal translation target for BPF. It is designed to be easier to compile to native code and, because of its ability to make external calls into the kernel, can easily be extended. eBPF programs can be compiled to native code or, if a compiler isn't available for a particular architecture, interpreted.

As of Linux 3.18, eBPF is available to user space programs[3]. It is the goal of some Linux developers to make eBPF the "universal in-kernel virtual machine"[7] replacing the handful of others that exist. There also is an LLVM backend so that application developers can write eBPF programs in C, rather than the raw eBPF bytecode.

eBPF is an attractive target for verification. One reason is its widespread and increasing use, as described above. If Linux is converging on one in-kernel virtual machine, having a verified compiler for that language would have a high impact. Any eBPF system, by design, allows the execution of user-supplied code on potentially user-supplied data within the kernel – so its security is critical. However, bugs and vulnerabilities have been found; Jitk describes many. Verification could prevent these bugs. As an example of complexity of the system, one part of the current Linux code that verifies the safety of eBPF programs is a 2003 line C file[8]. It is possible this verifier is bug free, but a proof of that would be useful.

2.3.3 Linux Verifier

The verifier for the Linux use of eBPF goes beyond simple checks on instructions (like no divide-by-immediate-zeros). For example, there are a number of restrictions placed on eBPF beyond those placed on BPF, such as preventing registers from being read before they are written and checking the types of arguments to external function calls. These occur statically, and allow for improved performance because the checks do not have to occur at runtime. This enables the simple instruction-to-instruction JIT goal of eBPF.

3 eJitk

This section describes the progress I made towards implementing eJitk, a verified compiler for eBPF based off of BPF. The design for eJitk is based off of Jitk's approach. Key features I added were the support for a type-checking filter, which can enforce the types of registers and maintain these types across loads and stores. I did not implement all of eBPF and did not create a full end-to-end compilation path.

The approach I took was to start from the existing BPF compiler and incrementally modify it to account for the differences between eBPF and BPF. This plan was feasible because of the fundamental similarity between BPF and eBPF.

3.1 Design

The design for the verified eBPF compiler is identical in structure to that of the existing BPF compiler in Jitk. The eBPF compiler is more complex due to the additional features, discussed in section 2.3.1, that eBPF provides compared to original BPF.

The eBPF verified compiler is written in Coq. It order to achieve eBPF $\rightarrow$ native translation, I built a eBPF $\rightarrow$ Cminor compiler. CompCert's Cminor $\rightarrow$ native translation would be used to finish the compilation from eBPF to native assembly. As in Jitk, there are five main components: the specification, the filter, the translation function, the semantic preservation proof, and the termination proof.

3.1.1 Specification

The specification provides three things. First, a datatype representing the possible states that a eBPF computation can be in. Second, an enumeration of the eBPF opcodes. Third, the semantics of eBPF – what effect each opcode has on the state and under what conditions an opcode is defined.

3.1.2 Filter

The filter is a function that statically checks an eBPF program (a list of eBPF opcodes) for conformance with the specification. Every opcode in a program for which the filter returns true must have defined behavior. Some things the filter checks for include negative jump offsets and division by zero. I added a second filter that ensures that registers are properly typed.

3.1.3 Translation Function

The translation function is the actual JIT compiler. This is a Coq function that translates an eBPF program into a Cminor AST that preserves the semantics of the eBPF program.

3.1.4 Semantic Preservation Proof

This is a proof that the translation function preserves the semantics of the provided eBPF. It relies on a matching function from eBPF states to Cminor states.

3.1.5 Termination Proof

The termination proof verifies that if the filter accepts an eBPF program then that program eventually terminates. This proof also ensures that the filter does not allow any undefined behavior.

3.2 Typed Register File

The typed register file is the primary contribution of this project. As previously described, the Linux verifier keeps track of the types that each register has. These types are used to statically ensure certain properties of an eBPF program. Implementing this required modifying the eBPF specification, the filter, and the termination proof.

I implemented three register types:

- `RCInt`, an integer type
- `RCPointer RPframe`, a frame pointer to the start of the stack
- `RCPointer RPctx`, a context pointer to the start of the input data

The register file itself is a mapping from a register number (0–10) to an optional register content type. The register file can be empty for a register—that means that that register's contents are undefined.

The semantics of eBPF instructions were changed to include preconditions on the types of values expected to be in the provided registers. For example, the `exit` instruction requires, as specified, that the contents of register 0 be defined and an `RCInt`. The new filter enforces this and the termination proof proves that the filter does so.

I added a new filter function that can statically type check registers by recursively walking every execution path. This is similar in functionality to the verifier in the Linux kernel. It uses an abstract *register file state* to keep track, at each recursive invocation, of what types are in each registers. When it checks an instruction that modifies a register, it changes the register file state that it passes to the recursive call. Conditional jumps are handled by validating both branches of the jump.

The termination proof states that if this filter function accepts an input, then the eBPF code will be fully defined by the specification, meaning that any preconditions on register types will be satisfied. It does that using a universal quantifier on register file states passed into the filter function, then specifying that universal quantifier to match the *actual* register file of some computation.

3.3 Basic Features

Using the typed register file, some basic features of eBPF were straightforward to implement. ALU operations can address any registers, but the register used as a source must be a defined integer. The `exit` instruction ensures that register 0 has a value, so it is not possible to run an eBPF program that returns an undefined value. Another eBPF change was to make jumps only have one target, which was straightforward to implement.

3.4 Loads and Stores

Loads and stores were a complicated eBPF feature that I implemented. They also relied on the typed register file. There are two types of loads in eBPF. The first are the backwards compatible loads from the input packet. These load instructions have a complicated specification: register 6 must contain a `RCPointer RPframe`, the result is returned in register 0, and registers 1–5 are *scratched*, or set to undefined, because this load is actually implemented in Linux as a function call. I was able to replicate this specification in eBPF, including the scratching of registers and requirement on register 6.

The second type of load has a corresponding store instruction. They are *generic* loads and stores, which take a source register, an offset, and a destination register as arguments.

Load `R[dst] = *(R[src] + off)`

Store `*(R[dst] + off) = R[src]`

The base register (destination for store and load for source) must be a `RCPointer` type, which can be enforced using the typed register file. Depending on the type of the pointer, these loads and stores do very different things.

If the pointer is a `RPctx`, the loads and stores are straightforward, either loading or storing integers from the input packet. The stack loads and stores, when the base register is a `RPframe` pointer, are more interesting, because the types of registers are preserved across stack spill–fills. The stack is implemented in the eBPF state as a map from stack-slots to an optional register content type. When doing a load or store to stack memory, the offset is converted into an index into this map and used to properly update the eBPF state.

4 Remaining Work

I did not completely implement the full eBPF language or the full Jitk process for eBPF. An important language feature not implemented is eBPF's ability to make external calls. These calls are typed check by the Linux verifier against a whitelist of allowed calls and what values must be in registers 1–5.

This would be possible to implement in eBPF and would require extending the specification to perform the proper type checking and generate the needed trace for the refinement proof.

Parts of the Jitk process not in eJitk are encoding / decoding and extraction. Jitk has a way to encode a BPF program to bytes and decode it back into abstract objects. There is an automatic proof that shows that the encode–decode preserves the structure. Jitk also has extraction, which enables BPF code compiled using it to actually be run. The eBPF compiler in eJitk has no way of actually running.

5 Reflection

This section briefly reflects on my experience performing proof engineering in Coq.

Incremental Approach Worked The approach I took was to modify Jitk, incrementally morphing it into eJitk aiming to have working proofs at each step. This approach worked well. It allowed me to experiment and add new features with the goal of doing the proof, not building the infrastructure around the proof. Also, returning to a working proof was always a `git stash` away. This approach allowed me to build off of others code which was a good learning experience. This was also a downside of the approach—I emulated much of the structure and style of the existing work, rather than always thinking deeply about what the best way to do something was.

Math is Hard Let *annoyed-hours* be a metric computed by multiplying the time a proof takes to prove by the amount of frustration suffered during that proof. Math proofs were above and beyond the largest source of annoyed-hours. For example, early in the project, I spent nearly four hours proving `0 <= 4 < Int.max_unsigned`, which is obviously and actually true. These hours were nearly constant frustration, as it was obvious that my goal was true, but I did not have the mental tools to prove it. I became more proficient at these sorts of proofs as I gained experience, but some things still seemed disproportionately hard. I could imagine these sorts of proofs being a large impediment to other engineers attempting to learn Coq. Better or more prominently documented automation tactics could help.

No Extraction Another frustrating part of Coq engineering that took time to get used to was never running code, especially compared to dynamically typed interpreted languages, such as Python, where "unit tests are the type system". The Coq code never has to run, just compile. I never implemented extraction for eJitk, so it is not even *possible* at the moment to run the code I was working on. I see the value to this—it is proved correct so there is no need to run it. But I also sense that it could have a negative aspect, where the real world requirements for systems could be shuffled to the back of the mind as engineers focus on getting proofs to pass.

6 Conclusion

This paper presented the background, design, and discussion of my 6.888 project, eJitk. The project made good progress in implementing a version of Jitk for eBPF. The responsibilities of the Linux eBPF system are expanding. Having a verified compiler for eBPF would greatly reduce the likelihood of bugs in a complicated and security critical part of the kernel.

7 References

[1] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock, "Jitk: a trustworthy in-kernel interpreter infrastructure," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pp. 33–47, USENIX, 2014.

[2] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference*, pp. 2–2, USENIX Association, 1993.

[3] J. Corbet, "Attaching eBPF programs to sockets." `http://lwn.net/Articles/625224/`.

[4] J. Corbet, "A JIT for packet filters." `http://lwn.net/Articles/437981/`.

[5] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[6] J. Schulist, D. Borkman, and A. Starovoitov, "Linux socket filtering aka berkeley packet filter (BPF)." `https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/Documentation/networking/filter.txt?id=refs/tags/v3.19.2`. Linux Source Code Documentation.

[7] J. Corbet, "BPF: the universal in-kernel virtual machine." `http://lwn.net/Articles/599755/`.

[8] Linux Kernel, "kernel/bpf/verifier.c." `https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/kernel/bpf/verifier.c?id=refs/tags/v3.19.2`.