Contribution ID: **133** Type: **not specified**

A proof-carrying approach to building correct and flexible BPF verifiers

Thursday, September 23, 2021 9:30 AM (40 minutes)

The BPF verifier is an integral part of the BPF ecosystem, aiming to prevent unsafe BPF programs from executing in the kernel. Due to its complexity, the verifier is susceptible to bugs that can allow malicious BPF programs through. A number of bugs have been found in the BPF verifier, some of which have led to CVEs (1, 2, 3). These bugs are severe, since the verifier is on the critical path for ensuring kernel security.

Due to its design, the verifier is also overly strict: it may reject many safe BPF programs because it lacks sophisticated analyses to recognize their safety. When a BPF program is rejected by the verifier, it can be a frustrating experience (4). To get their program accepted by the verifier, developers often have to resort to ad-hoc fixes, tweaking C source code or disabling optimizations in LLVM. This solution becomes brittle as developers write more complex BPF programs and new optimizations are introduced in LLVM.

In this talk, we argue that a more systematic approach is to freeze the kernel side of the BPF verifier and move most of its complexity to user space. To do so, we introduce formal, machine-checkable proofs of the safety of BPF programs. Applications provide proofs that their BPF programs are safe, and a proof checker in the kernel validates the proofs. By decoupling proof validation from generation, this achieves two goals. First, the kernel side of the interface is fixed to be a specification of BPF program safety and the proof checker, avoiding the ever-growing complexity of the BPF verifier in the kernel. Second, applications can choose an appropriate strategy to generate proofs for their BPF programs. Since the proofs are untrusted, there is no risk of applications introducing bugs from complex proof strategies.

We have been building a prototype BPF verifier using this approach. Our prototype uses the logic of the Lean theorem prover (5), which has been thoroughly analyzed (6) and has multiple independent implementations of proof checkers (7). We are developing two automated strategies for generating proofs. The first strategy mimics the current BPF verifier. It implements an abstract interpreter for BPF programs that uses ranges and tristate numbers to approximate sets of values of BPF registers. The second strategy uses symbolic execution to encode the semantics of a BPF program as boolean constraints, which are discharged using a SAT solver. Both strategies produce proofs that are validated by the proof checker, avoiding the possibility of introducing bugs like those that have been found in the current verifier.

Our goal is to present an alternative approach to building the BPF verifier, and explore the advantages and limitations of this approach. We would like to start a discussion on ways to combine both approaches in a pragmatic way.

I agree to abide by the anti-harassment policy

I agree

Primary authors: NELSON, Luke (University of Washington); WANG, Xi (University of Washington); TOR-

LAK, Emina (University of Washington)

Presenters: NELSON, Luke (University of Washington); WANG, Xi (University of Washington); TORLAK, Emina (University of Washington)

Session Classification: BPF & Networking Summit

Track Classification: Networking & BPF Summit (Closed)