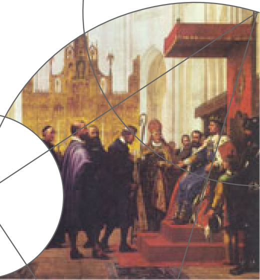
Faculty of Science

Proof checking in the Linux kernel

Jacob Herbst (mwr148)
Institute of Computer Science (DIKU)



Agenda

- eBPF, Proof Carrying Code and why we need a proof-checker in the kernel
- Logical Framework with Side Conditions (LFSC)
- An LF proof
- Shortcomings of the implementation
- Conclusion

eBPF and why we want formal verification

- eBPF is a virtual machine that gives sandbox functionality with kernel privileges.
- limbo between usability and safety
- programs are checked by static analysis called the eBPF verifier.
  - 19k lines of code.
  - many bugs, making eBPF security hazard
  - disabled by default in most Linux distributions.

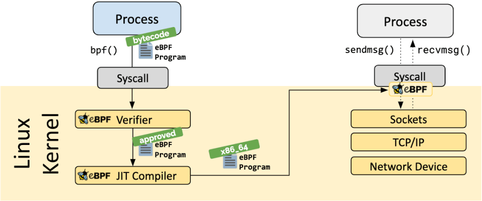


Figure: Loading process of eBPF[1]

[1] https://ebpf.io/what-is-ebpf/

Proof Carrying Code

- code producer is responsible for ensuring the correctness
- code consumer shall check that proofs have not been tampered and that a certificate is valid.

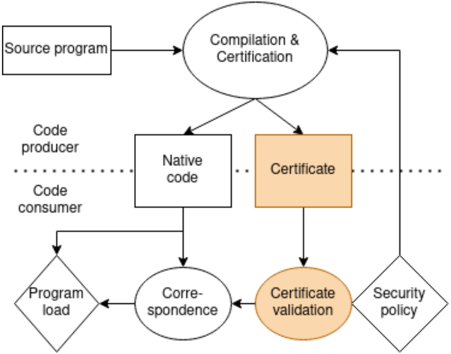


Figure: Structure of PCC

Logical Framework - with Side Conditions?

- An extension of the simply typed lambda calculus with dependent product type
  - $\prod_{x:A} B(x)$, with $A : \mathcal{U}$ and $B : A \to \mathcal{U}$
  - if $x \notin FV(B)$ then $A \to B$
- Used for computer-assisted proofs using the Curry-Howard correspondence

Formal Logic	Type Theory
$\top$ (true)	()
$\bot$ (false)	void
$\wedge$ (conjunction)	product type
$\vee$ (disjunction)	sum type
$\to$ (implication)	function type
$\forall$ (universal quantification)	Π(dependent type)

- LFI extend LF with implicit arguments
- LFSC extend LFI with Side Conditions

LFSC syntax

$$
\begin{aligned}
K &::= \mathbf{type} \mid \mathbf{type}^c \mid \mathbf{kind} \mid \Pi x : A.K \mid \mathbf{int} \mid \mathbf{rational} \\
A, B &::= a \mid A\ M \mid \Pi x : \{S\ M\}.A \mid \Pi x : A.B \\
M, N, O &::= x \mid c \mid z \mid q \mid * \mid M : A \mid \mathbf{let}\ x\ M\ N \mid \lambda x.M \mid \lambda x : A.M \mid M\ N \\
P &::= c \mid c\ x_1 \ldots x_n \\
S, T, U &::= x \mid c \mid -S \mid S \oplus S \mid c\ S_1 \ldots S_n \mid \mathbf{let}\ x\ S\ T \mid \mathbf{markvar}\ S \mid \\
&\quad\ \mathbf{ifequal}\ S_1\ S_2\ T\ U \mid \mathbf{match}\ S\ (P_1\ T_1)\ \ldots\ (P_n\ T_n) \mid \mathbf{fail}\ S \mid \\
&\quad\ \mathbf{ifneg}\ S\ T\ U \mid \mathbf{ifzero}\ S\ T\ U \mid \mathbf{ifmarked}\ S\ T\ U \mid \mathbf{ztoq}\ S \\
\oplus &\in \{+, /, *\cdot\}
\end{aligned}
$$

Figure: Syntactical categories of LFSC

LFSC typing

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{type} \Rightarrow \mathbf{kind}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{int} \Rightarrow \mathbf{type}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{rational} \Rightarrow \mathbf{type}}$$

$$\frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \frac{\vdash \Gamma \quad a : K \in \Sigma}{\Gamma \vdash a \Rightarrow K} \qquad \frac{\vdash \Gamma \quad c : A \in \Sigma}{\Gamma \vdash c \Rightarrow A}$$

$$\frac{\Gamma \vdash_\Sigma A \Leftarrow \mathbf{type} \quad \Gamma, x : A \vdash_\Sigma C \Rightarrow \alpha \quad \alpha \in \{\mathbf{type}, \mathbf{type}^{\mathbf{c}}, \mathbf{kind}\}}{\Gamma \vdash_\Sigma \Pi x : A.C \Rightarrow \alpha}$$

$$\frac{\Gamma \vdash_\Sigma A \Rightarrow \Pi x : B.\ K \quad \Gamma \vdash_\Sigma M \Leftarrow B}{\Gamma \vdash_\Sigma A\ M \Rightarrow [M/x]K} \qquad \frac{\Gamma \vdash_\Sigma M \Rightarrow \Pi x : A.\ B}{\Gamma \vdash_\Sigma M * \Rightarrow [*/x]B}$$

$$\frac{\Gamma \vdash_\Sigma M \Rightarrow \Pi x : A.\ B \quad \Gamma \vdash_\Sigma N \Leftarrow A}{\Gamma \vdash_\Sigma M\ N \Rightarrow [N/x]B} \qquad \frac{\Gamma \vdash_\Sigma M \Leftarrow A}{\Gamma \vdash_\Sigma M : A \Rightarrow A}$$

$$\frac{\Gamma \vdash_\Sigma A \Rightarrow \mathbf{type} \quad \Gamma, x : A \vdash_\Sigma M \Rightarrow B}{\Gamma \vdash_\Sigma \lambda x : A.\ M \Rightarrow \Pi x : A.\ B} \qquad \frac{\vdash \Gamma}{\Gamma \vdash_\Sigma q \Rightarrow \mathbf{rational}}$$

$$\frac{\Gamma, x : A \vdash_\Sigma M \Rightarrow B}{\Gamma \vdash_\Sigma \lambda x.\ M \Leftarrow \Pi x : A.\ B} \qquad \frac{\vdash \Gamma}{\Gamma \vdash z \Rightarrow \mathbf{integer}}$$

Figure: Bidirectional typing rules for LFSC

$2+2=4$ using cvc5 and LFSC

```
1  (check
2  (@ t1 (int 4)
3  (@ t2 (int 2)
4  (@ t3 (= (a.+ t2 (a.+ t2 (int 0))) t1)
5  (# a0 (holds (not t3))
6  (: (holds false)
7  (eq_resolve _  _  a0
8  (trans _  _  _
9  (cong _  _  _  _
10 (refl f_not)
11 (trans _  _  _
12 (cong _  _  _  _
13 (cong _  _  _  _
14 (refl f_=)
15 (trust t3))
16 (refl t1))
17 (trust (= (= t1 t1) true))))
18 (trust (= (not true) false)))))))))
```

$$\text{eq\_resolve} : \prod_{f:T}\prod_{g:T}\prod_{p_1:[f]}\prod_{p_2:[f=g]}[g]$$

$$\text{trans} : \prod_{t_1:T}\prod_{t_2:T}\prod_{t_2:T}\prod_{u_1:[t_1=t_2]}\prod_{u_2:[t_1=t_3]}[t_1=t_3]$$

$$\text{cong} : \prod_{a_1:T}\prod_{b_1:T}\prod_{a_2:T}\prod_{b_2:T}\prod_{u_1:[a_1=b_1]}\prod_{u_2:[a_2=b_2]}[a_1\,a_2=b_1\,b_2]$$

$$\text{refl} : \prod_{t:T}[t=t]$$

$$\text{holds} : \prod_{t:T}T$$

$$= \stackrel{\text{def}}{=} \lambda t_1 : T.\lambda t_2 : T.f_{=}\,t_1\,t_1$$

$$\dfrac{\dfrac{}{\neg(2+2=4)}\,a_0\,(5) \quad \dfrac{\dfrac{\dfrac{\neg}{\neg=\neg}\,\text{refl (10)} \quad \dfrac{\dfrac{\dfrac{\dfrac{f_{=}}{(=)=(=)}\,\text{refl (14)} \quad \dfrac{t_3}{(2+2=4)}\,\text{trust (15)}}{(2+2=)=(4=)}\,\text{cong (13)} \quad \dfrac{t1}{4=4}\,\text{refl (16)}}{(2+2=4)=(4=4)}\,\text{cong (12)} \quad \dfrac{(4=4)=\top}{(4=4)=\top}\,\text{trust (17)}}{(2+2=4)=\top}\,\text{trans (11)}}{\neg(2+2=4)=\neg\top}\,\text{cong (9)} \quad \dfrac{\neg\top=\bot}{\neg\top=\bot}\,\text{trust (18)}}{\neg(2+2=4)=\bot}\,\text{trans (8)}}{\bot}\,\text{eq\_resolve (7)}$$

Demo

Demo

Why can we also prove $2 + 3 = 4$?

Demo

Why can we also prove $2 + 3 = 4$?

```
(declare apply (! t1 term (! t2 term term)))
(declare f_a.+ term)
(define a.+ (# x term (# y term (apply (apply f_a.+ x) y))))
```

$$a.+ : (\lambda x : term.\lambda y : term.term)$$

Demo

Why can we also prove $2 + 3 = 4$?

```
(declare apply (! t1 term (! t2 term term)))

(declare f_a.+ term)

(define a.+ (# x term (# y term (apply (apply f_a.+ x) y))))
```

$$a.+ : (\lambda x : term.\lambda y : term.term)$$

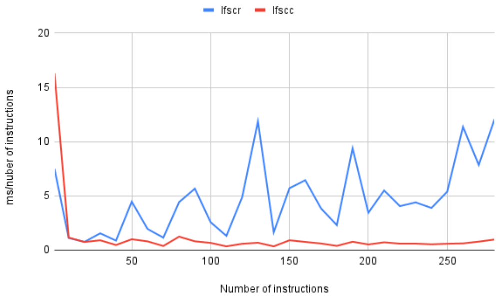
We could just as well use -, as the definition is as follows:

```
(declare f_a.- term)
(define a.- (# x term (# y term (apply (apply f_a.- x) y))))
```

This is a problem if we want to use it to make proofs by hand, but in a PCC context, it will suffice, as long as we are sure the in the kernel VC generator is sound.

Shortcommings

- Implementation in this thesis is not fast.
- Approach follows a typically staged compilation. Parser → Transformation → Typechecking.
- Overhead in both memory and runtime.
- *lfscc* takes an online approach, where parsing, and type checking happens all at once (no intermediate stages)



Conclusion

- Side Conditions might be unnecessary
- Formal verification of eBPF programs will take longer than the eBPF verifier but can be done fairly efficiently, but not with the approach tried in my work.
- Proposal is to take a similar approach to *lfscc* and implement it in Rust, as it seems like a doable solution.