

# Safe Kernel Extensions Without Run-Time Checking

Because run-time checking makes  
things really, really slow. 🙄

# What do we want?

- Third-party extensions to the kernel
  - Extend functionality
  - Device drivers
  - Often necessary – the kernel programmers can't cover everything
- However, this brings several concerns:
  - Safety – Don't crash me, bro!
  - Trust – Jim-Bob's Haus of Coding Good
  - Speed – Slow code makes users cry
- How do we satisfy these concerns?

# 😄 Proof Carrying Code! 😄

- Rules to enforce
  1. Don't hit your sister
  2. Don't run with scissors
- User / Consumer specifies policy
- Developer proves the code satisfies the policy
- User verifies proof, then runs module

# The stuff

- Deception-proof
- Very flexible – wide range of possible policies
- Excellent example of “Ensure vs. Check”
  - Trust is no longer an issue
- Very small trusted base (just the verifier)
- Runs native code without modification
  - *No performance penalty*

# Main difficulties

- Creating the proofs – “The trick is not minding that it hurts”
  - Certifying compilers
- Stating the safety policy
  - Requires logically representing the machine
- Size of the proof
  - Can be compressed

# The wind-up – how do we start?

1. Abstract machine specification
  - Basic representation
  - Agreed upon by Consumer and Producer
2. Axioms and Predicates
  - **rd(a), wr(a)**
3. Declaration of precondition *Pre*
  - “You, Producer, can make these assumptions”
4. Declaration of postcondition *Post*
  - “Producer, your code needs to meet these conditions when it finishes”

# Where the magic happens: Creating the proof

- Code Producer: take spec,  $Pre$ ,  $Post$ , and module program  $\Pi$
- Start at end and work backwards – obtain initial verification condition predicate  $VC_0$
- Add explicit invariants when needed
- Check that  $SP(\Pi, Pre, Post) = \forall \mathbf{r}_0 \dots \forall \mathbf{r}_{10} \forall \mathbf{r}_m. Pre \Rightarrow VC_0$  holds
- Distribute Code, Proof, and Mapping

# Back at the ranch: Checking the proof

- Use First-order Logic
  - Use a theorem prover – LF
- Progress lemma – Theorem 2.1
  - “We either finish, or we can take a step forward”
  - Violations of the policy can’t move forward
- Map proof to code, run through code + proof linearly using theorem prover

# A real-world example: Packet filters

- Hand-coded assembly
  - PCC works with all code
  - Optimizations? No problem!
- Requires fast execution
- Good case study
  - Several real-world memory safety reqs
  - Comparison with existing approaches

# Holy logic formulas, Batman!

$$\begin{aligned} Pre = & \mathbf{r}_1 \bmod 2^{64} = \mathbf{r}_1 \wedge \\ & \mathbf{r}_2 \bmod 2^{64} = \mathbf{r}_2 \wedge \mathbf{r}_2 < 2^{63} \wedge \mathbf{r}_2 \geq 64 \wedge \\ & \mathbf{r}_3 \bmod 2^{64} = \mathbf{r}_3 \wedge \\ & \forall i. (i \geq 0 \wedge i < \mathbf{r}_2 \wedge (i \& 7) = 0) \\ & \quad \Rightarrow \mathbf{rd}(\mathbf{r}_1 \oplus i) \quad \wedge \\ & \forall j. (j \geq 0 \wedge j < 16 \wedge (j \& 7) = 0) \\ & \quad \Rightarrow \mathbf{wr}(\mathbf{r}_3 \oplus j) \quad \wedge \\ & \forall i. \forall j. (i \geq 0 \wedge i < \mathbf{r}_2 \wedge j \geq 0 \wedge j < 16) \\ & \quad \Rightarrow (\mathbf{r}_1 \oplus i \neq \mathbf{r}_3 \oplus j) \end{aligned}$$

# What do we learn from that?

- No assumptions – state everything explicitly
- Initial idea of complexity involved

# A fancy graph with pretty lines

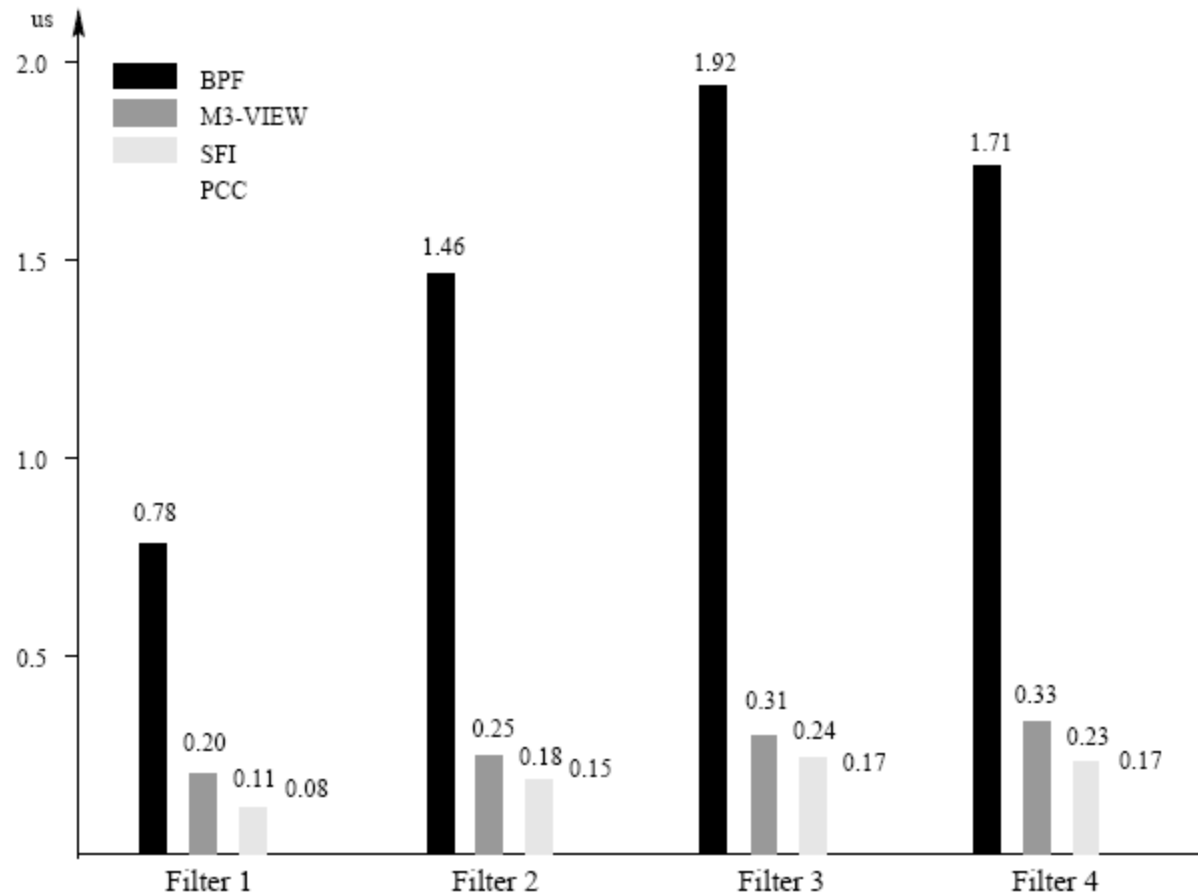


Figure 8: Comparison of average per-packet run time.

# Must go faster, must go faster...

- PCC wins on speed
  - Execution of native code (c.f. BPF)
  - Better understanding of code = more optimizations (c.f. Modula-3)
  - No run-time checks (c.f. SFI)
- Up-front cost with proof verification
  - Paid for within 60 seconds of execution

# Comparisons

- Code signatures
  - Good for trust
  - Not so good for safety – even gods make mistakes
- Java bytecode
  - Safe low-level language
  - Requires too many run-time checks
  - Only basic type information

# Comparisons (cont'd)

- SFI / CFI
  - Only memory safety
  - Requires modifying code
    - May change semantics
  - Run-time performance hit

# CFI + PCC?

- PCC can prove any safety predicate
- Put in CFI code – stop “powerful attackers”
- Use PCC to verify the code is tamper-proof
- Optimization of CFI – nix unnecessary checks

# Further thoughts

- Biggest concern: Automation of the proofs
  - Synthesizing proofs from code? (c.f. [Parent, 1995])
  - Synthesizing code from proofs? (c.f. Coq theorem prover)
  - Move towards declarative languages – compilation produces both?

# The Real World

- Proofs are necessary
- Average coder can't do them
- Want to prove other things – correctness
- Want wider applicability – beyond mission-critical applications

# I'd like to thank the Academy...

## (Citations)

- George C. Necula and Peter Lee. “Safe Kernel Extensions Without Run-time Checking.” In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, Seattle, Washington, October 1996.
- George Necula. Proof-Carrying Code. In Benjamin C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, MIT Press, Cambridge, MA, 2005.

... and great uncle Frank, and...  
(Citations, cont'd)

- Catherine Parent. Synthesizing proofs from programs in the calculus of inductive constructions. In *Proceedings of MPC'1995*, Volume 947 of *Lecture Notes in Computer Science*, pages 351-379, 1995.
- Yves Bertot, Pierre Casteran. *Interactive Theorem Proving and Program Development*, Springer, Berlin, 2004.