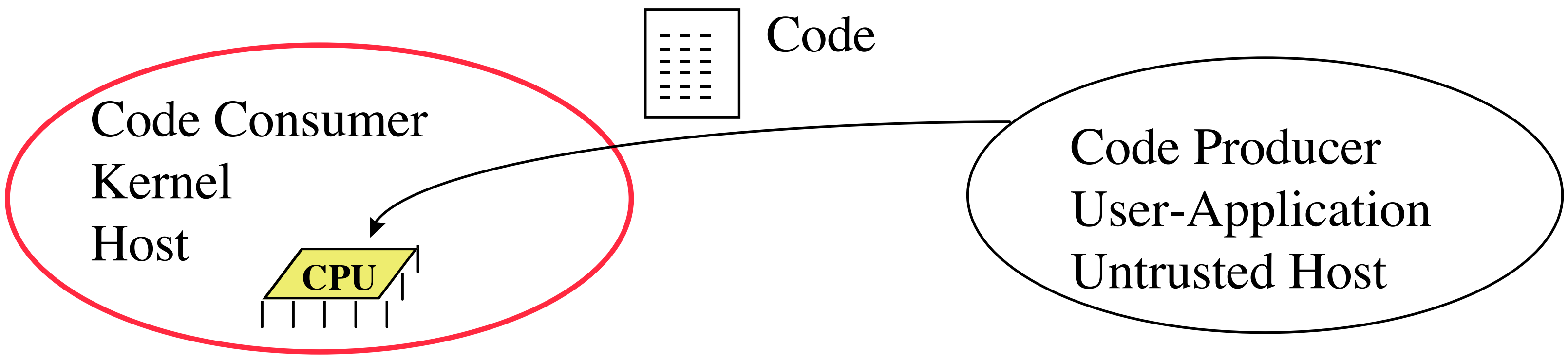
Safe Kernel Extensions without Run Time Checking

George C. Necula   Peter Lee

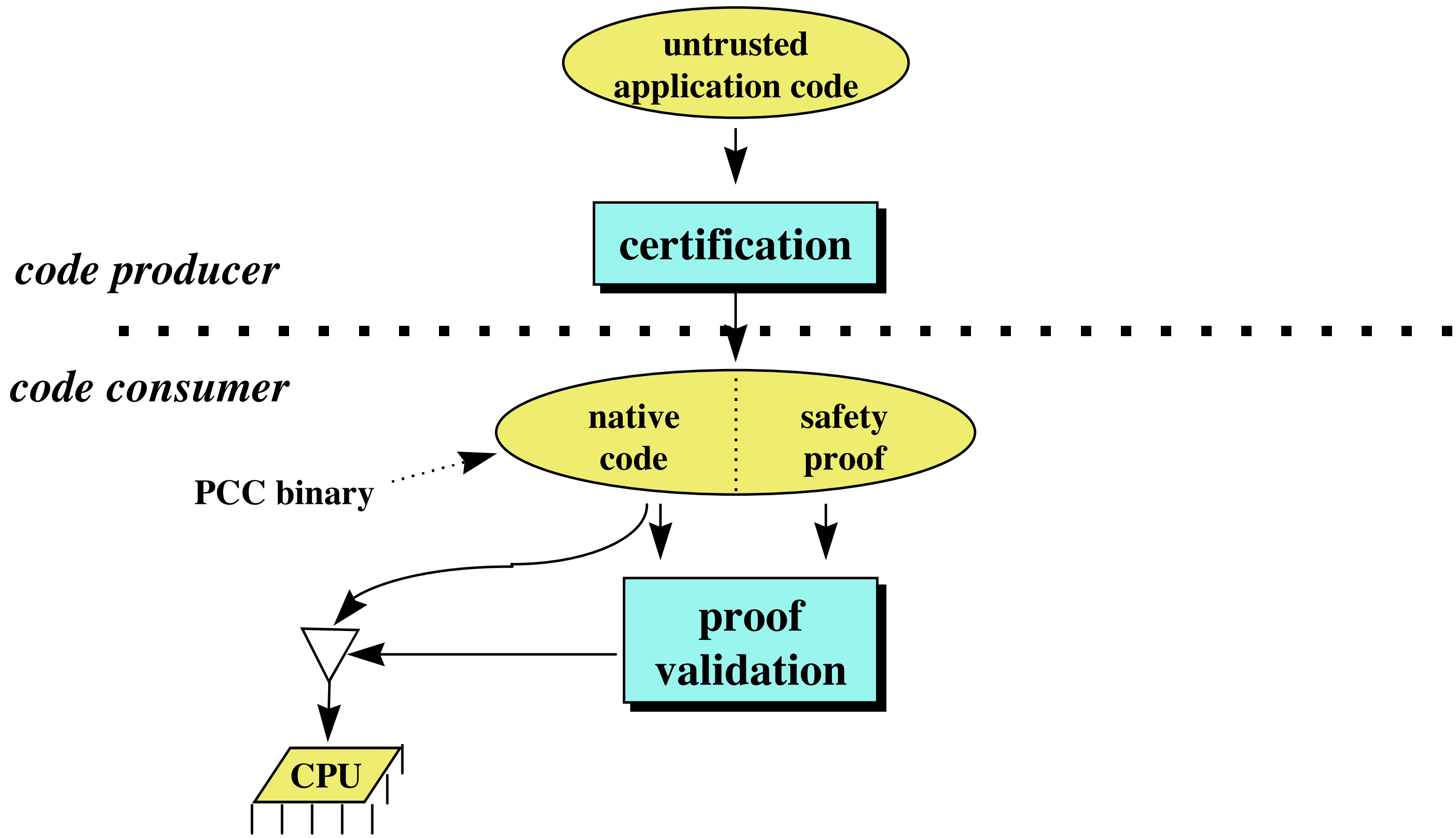
Carnegie Mellon University

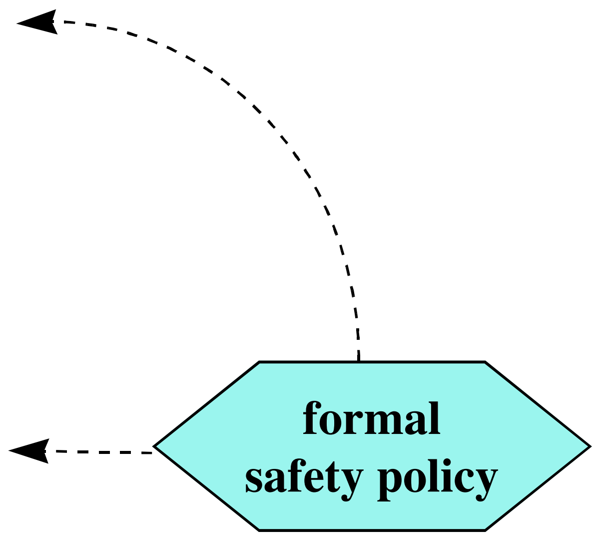
The Problem: Safety in the presence of untrusted code



- Examples: OS Extensions, Safe Mobile Code, Programming Language Interoperation
- Previous: Hardware memory protection, Runtime checking, Interpretation
- We want both safety and *performance!*

Proof-Carrying Code (PCC)

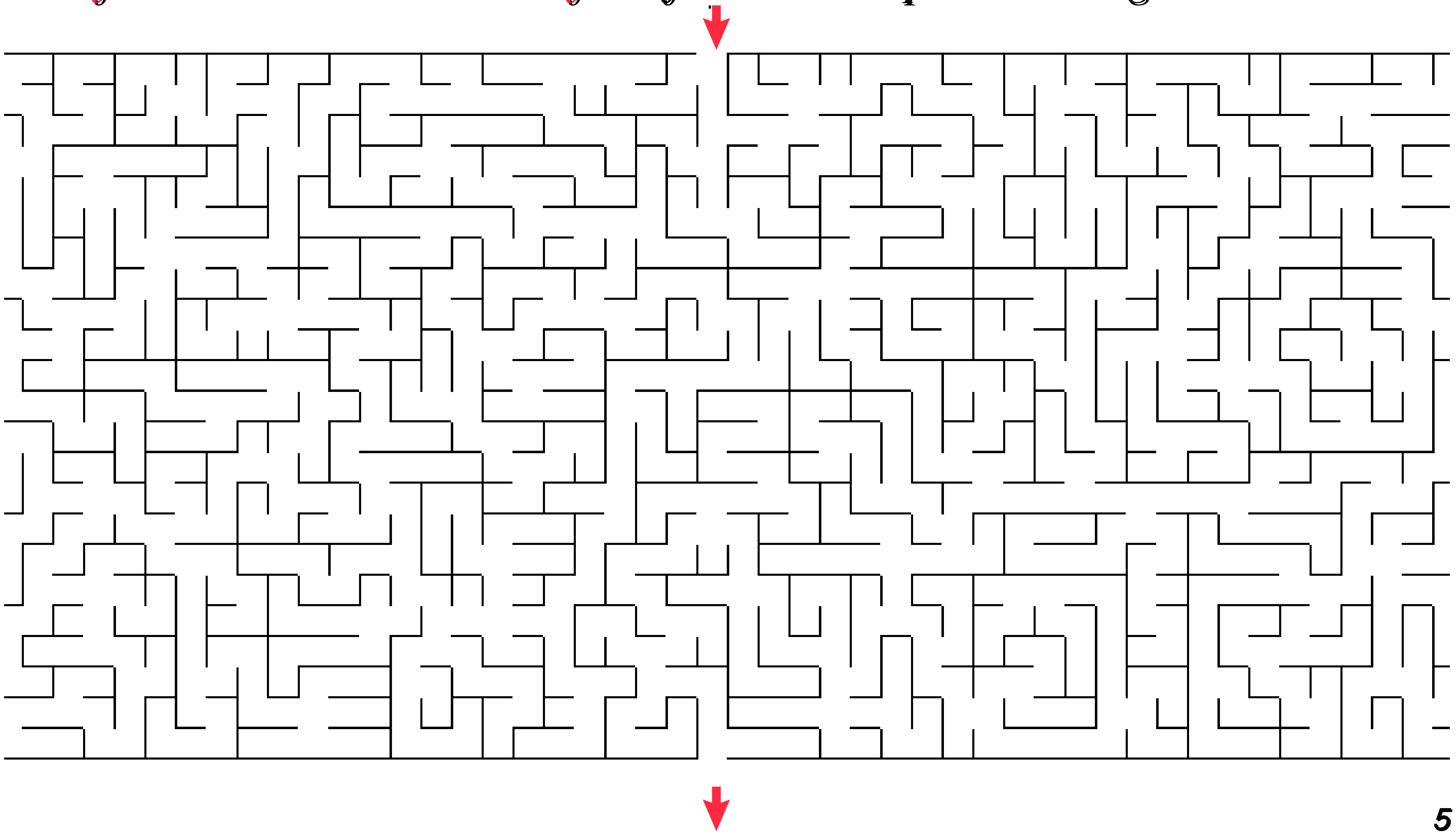


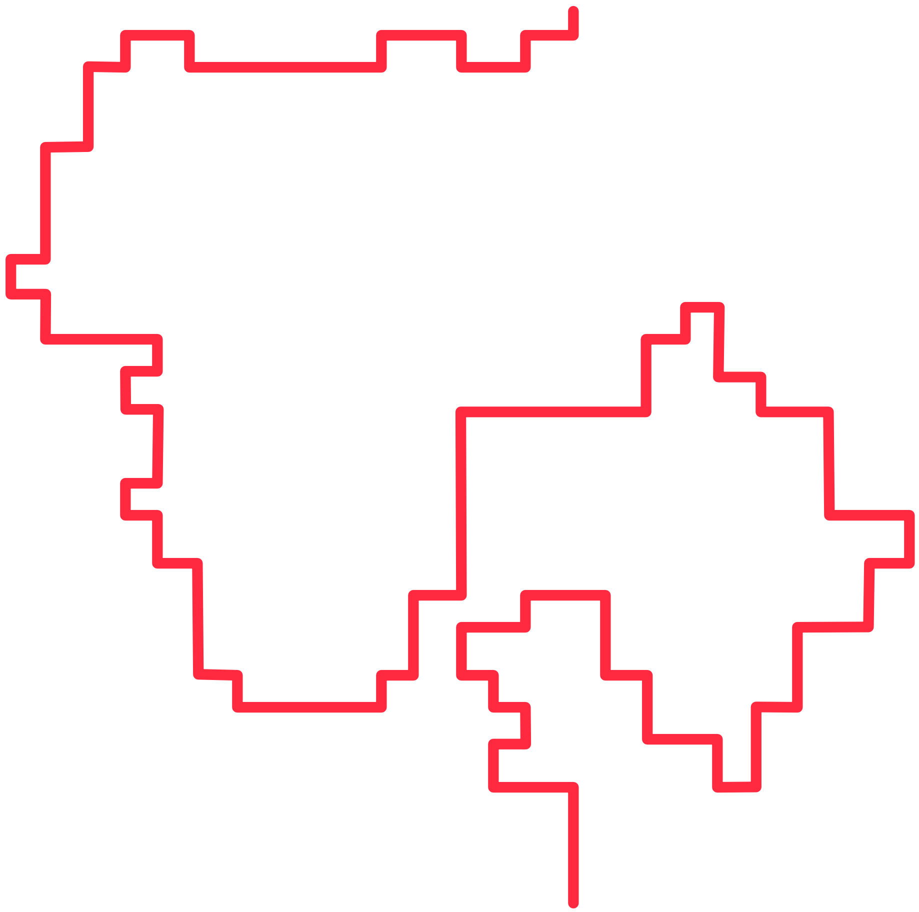

formal
safety policy

Checking a Proof vs. Generating One

Carnegie Mellon

***Definition*: A maze is "*safe*" if there is a path through it!**





Benefits of PCC

Carnegie Mellon

- Wide range of safety policies
  - ✔ memory safety
  - resource usage guarantees (CPU, locks, etc.)
  - concurrency properties
  - ✔ data abstraction boundaries
- Wide range of languages
  - ✔ assembly languages
  - high-level languages
- Simple, fast, easy-to-trust validation
- Tamper-proof

Experimentation

Carnegie Mellon

- Goal:
  - Test feasibility of PCC concept
  - Measure costs (proof size and validation time)
- Choose simple but practical applications
  - ⇨ Network Packet Filters
  - IP Checksum
  - Extensions to the TIL run-time system for Standard ML

Experimentation (2)

Carnegie Mellon

- Use DEC Alpha assembly language (hand-optimized for speed)
- Network Packet Filters
  - BPF safety policy: *"The packet is read-only and the scratch memory is read-write. No backward branches. Only aligned memory accesses."*

PCC Implementation (1)

- Formalize the safety policy:
  - Use first-order predicate logic extended with **can_rd(addr)** and **can_wr(addr)**
  - Kernel specifies safety preconditions
    - Calling convention
    - Guaranteed by the kernel to hold on entry

$$\forall i.(i \geq 0 \wedge i < r_1 \wedge i \bmod 8 = 0) \Rightarrow \text{can\_rd}(r_0 + i)$$

$$\forall j.(j \geq 0 \wedge j < 16 \wedge j \bmod 8 = 0) \Rightarrow \text{can\_wr}(r_2 + j)$$

PCC Implementation (2)

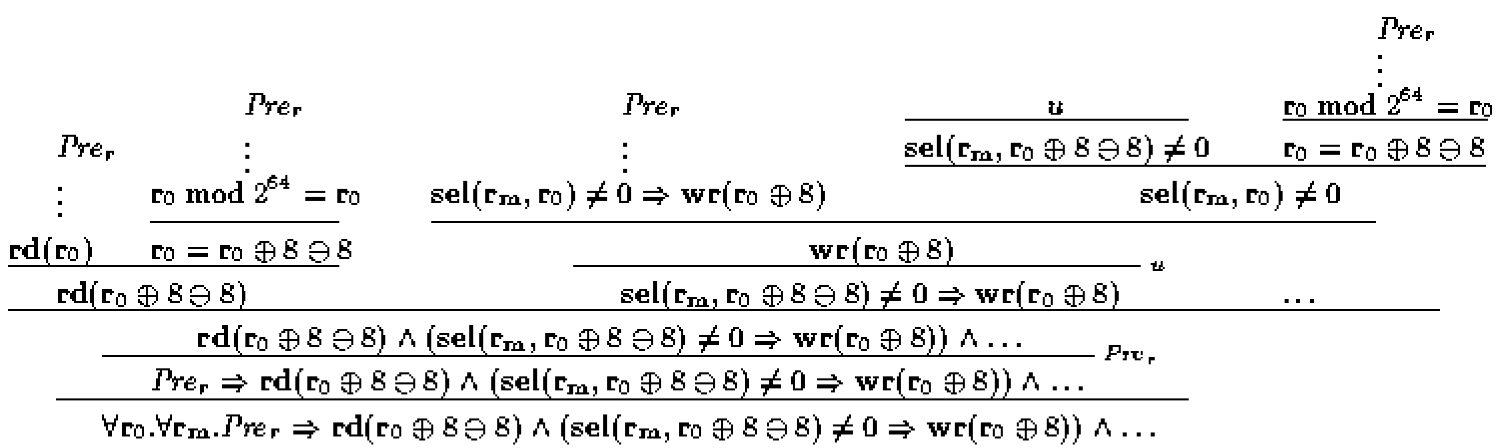
Carnegie Mellon

- Compute a safety predicate for the code
  - Use Floyd-style verification conditions (VCgen)
  - One pass through the code, for example:
    - For each **LD r,n[r_b]** add can_rd(r_b+n)
    - For each **ST r,n[r_b]** add can_wr(r_b+n)
- Prove the safety predicate
  - Use a general purpose theorem prover

Carnegie Mellon

PCC Implementation (3)

- Formal proofs are trees:
  - the leaves are axiom instances
  - the internal nodes are inference rule instances
  - at the root is the proved predicate
  - Example:



PCC Implementation (4)

Carnegie Mellon

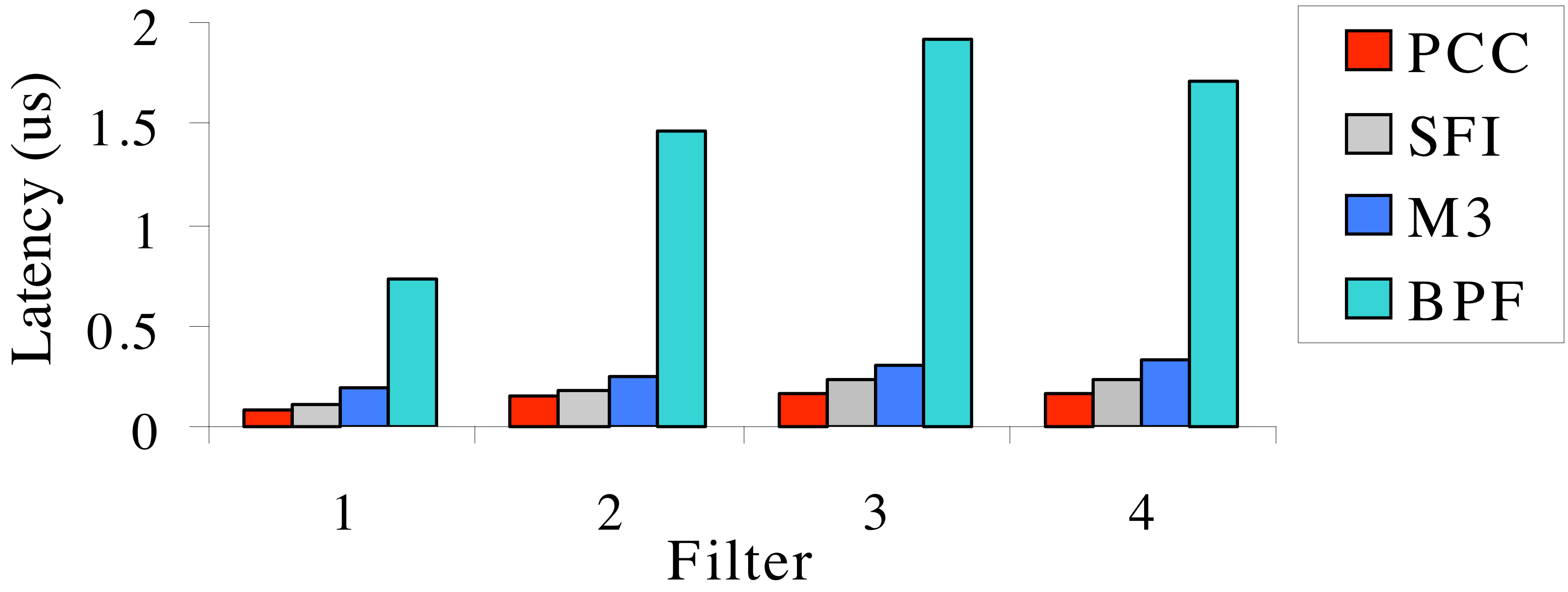
- Proof Representation: Edinburgh Logical Framework (LF)
- Proofs encoded as LF expressions
- Proof Checking is LF type checking
  - simple, fast and easy-to-trust (14 rules)
  - 5 pages of C code
  - independent of the safety policy or application
  - based on well-established results from type-theory and logic
- Large design space, not yet explored

Packet Filter Experiments

Carnegie Mellon

- 4 assembly language packet filters (hand-optimized for speed):
  1. Accepts IP packets (8 instr.)
  2. Accepts IP packets for 128.2.206 (15 instr.)
  3. IP or ARP between 128.2.206 and 128.2.209
  4. TCP/IP packets for FTP (28 instr.)
- Compared with:
  - Run-Time Checking: Software Fault Isolation
  - Safe Language: Modula-3
  - Interpretation: Berkeley Packet Filter

Performance Comparison



- Off-line packet trace on a DEC Alpha 175MHz
- PCC packet filters: fastest possible on the architecture
- The point: Safety without sacrificing performance!

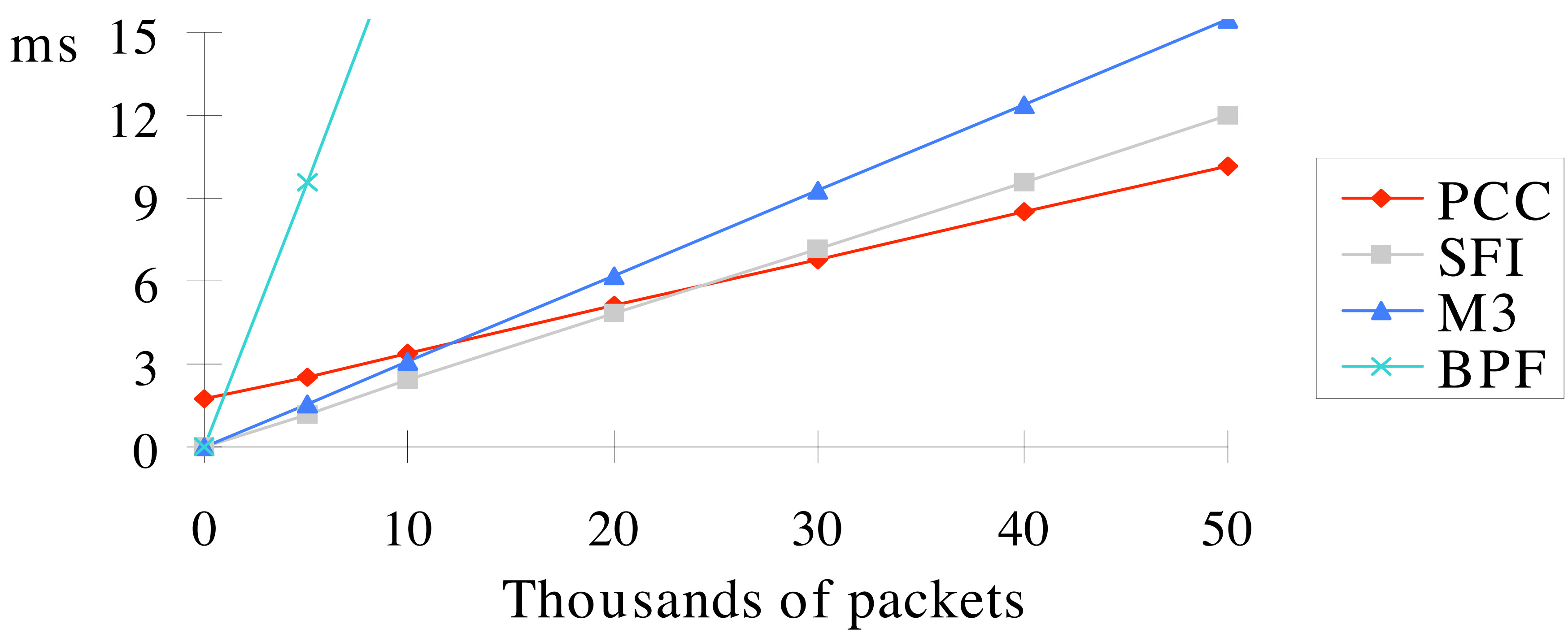
Cost of PCC for Packet Filters

Carnegie Mellon

- Proofs are approx. 3 times larger than the code
- Validation time: 0.3-1.8ms

Packet Filter	1	2	3	4
Instructions	8	15	47	28
Proof Size(bytes)	160	225	532	420
Validation Time(us)	362	872	1769	1354

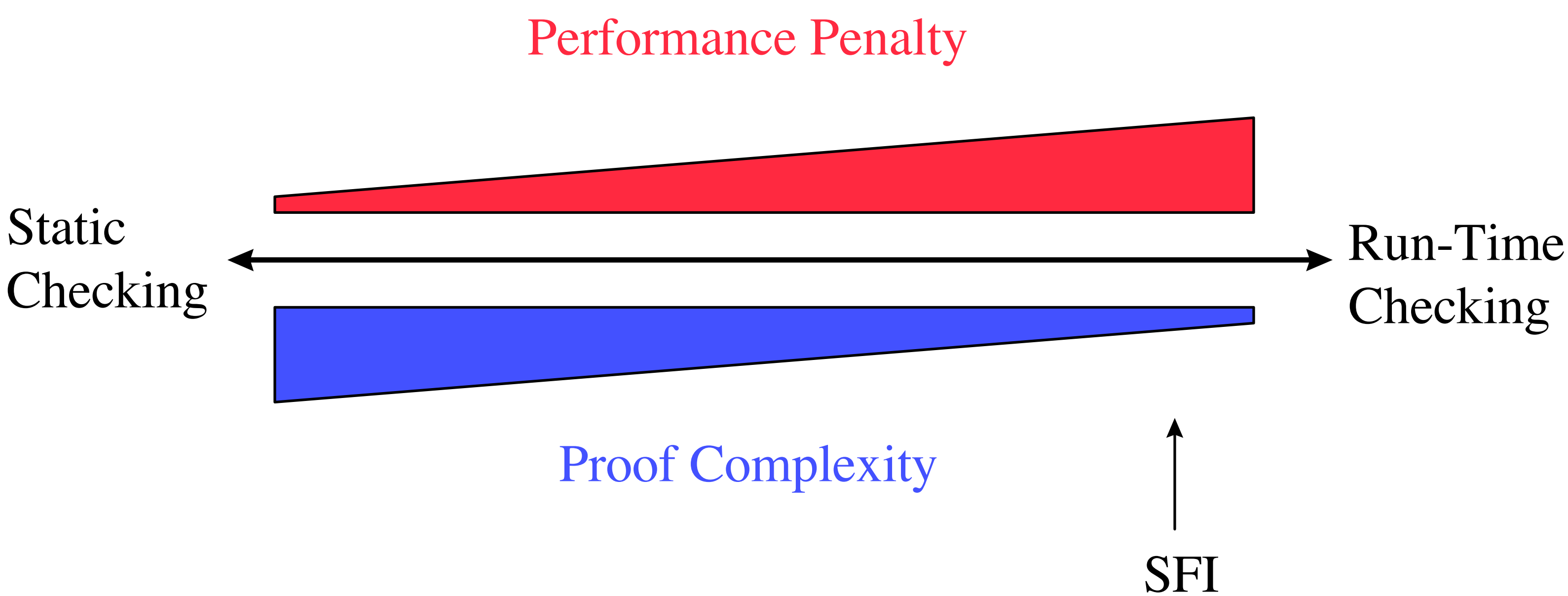
Validation Cost (Filter 3)



- Conclusion: One-time validation cost amortized quickly

PCC for Memory Safety

- Continuum of choices between static checking and run-time checking:



- PCC can be also used where run-time checking cannot (e.g., concurrency)

Practical Difficulties

- Proof generation
  - Similar to program verification
  - But:
    - done off-line
    - can use run-time checks to simplify the proofs
  - In restricted cases it is feasible (even automatable)
- Proof-size explosion
  - It is exponential in the worst case
  - Not a problem in our experiments

Future Work

- Resource Usage Policies
  - Locks, deadlock avoidance
- Certifying Compiler
  - Avoids theorem proving
  - Generates proof of type-safety for target code completely automatically
  - The most promising path towards large scale PCC
- More applications
  - Smartcards
  - Active Networks

Conclusion

Carnegie Mellon

- A very promising framework for ensuring safety of untrusted code.
- Achieves safety without sacrificing performance
- Type-safety properties for assembly language
- Serious difficulties exist
- Need more experimentation