Exokernel and Safe Kernel Extensions

Phil Gibbons

15-712 F15

Lecture 17

Today's Reminders

- **Midterm Grades**
  - Very preliminary: based only on summaries & class participation to date (only 10% of final grade points)
  - Kevin graded (check+, check, check-) ALL the summaries
- **Funds exist for time on AWS**
  - If your project could use AWS, say so in project proposal
- **Project proposals due Friday midnight**
- **My office hours: Today after class**

Safe Kernel Extensions without Run-Time Checking [OSDI'96]

- **George Necula** (UC Berkeley)
  - ACM Grace Hopper Award; "Test of Time" award for POPL'97 & POPL'02 papers

- **Peter Lee** (Microsoft)
  - ACM Fellow; Former CMU CSD Dept Head; "Test of Time" award for POPL'96 & POPL'97 papers

SigOps Hall of Fame paper inducted 2006

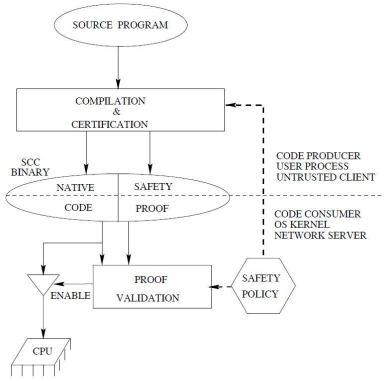
Safe Kernel Extensions without Run-Time Checking

"This paper introduced the notion of proof carrying code (PCC) and showed how it could be used for ensuring safe execution by kernel extensions without incurring run-time overhead. PCC turns out to be a general approach for relocating trust in a system; trust is gained in a component by trusting a proof checker (and using it to check a proof the component behaves as expected) rather than trusting the component per se. PCC has become one of the cornerstones of language-based security." – SigOps HoF citation

Overview

- **Mechanism for an OS kernel to determine with certainty that it is safe to execute a binary from an untrusted source**
- **"Proof-carrying code"**
  - Binary contains a formal proof that the code obeys the kernel's published safety policy
  - Kernel validates proof w/o using cryptography or consulting any external trusted entity
- **Main advantage: No run-time checking**
- **Main practical difficulty: Generating the safety proofs**

Overview of Proof-Carrying Code



Defining a Safety Policy

"The impatient reader may want to skip ahead to Section 3"

- **Floyd-style verification-condition generator**
  - Procedure that computes a "safety predicate" in first-order logic based on the code to be certified
  - Obtained by first specifying an "abstract machine" (operational semantics) that simulates the execution of safe programs
- **Set of axioms used to validate the safety predicate**
- **Precondition: "calling convention" for kernel to invoke PCC binary**

Abstract Machine for Memory-Safe DEC Alpha Machine Code

$$
\begin{array}{lcl}
op & ::= & n \mid \mathbf{r}_i \qquad i \in 0 \ldots 10 \\
al & ::= & \text{ADDQ} \mid \text{SUBQ} \mid \text{AND} \mid \text{OR} \mid \text{SLL} \mid \text{SRL} \\
br & ::= & \text{BEQ} \mid \text{BNE} \mid \text{BGE} \mid \text{BLT} \\
instr & ::= & \text{LDQ } \mathbf{r}_d, n(\mathbf{r}_s) \mid \text{STQ } \mathbf{r}_s, n(\mathbf{r}_d) \mid al\ \mathbf{r}_s, op, \mathbf{r}_d \mid br\ \mathbf{r}_s, n \mid \text{RET}
\end{array}
$$

$$
(\rho, pc) \rightarrow \begin{cases}
(\rho[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op], pc+1), & \text{if } \Pi_{pc} = \text{ADDQ } \mathbf{r}_s, op, \mathbf{r}_d \\
(\rho[\mathbf{r}_d \leftarrow \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)], pc+1), & \text{if } \Pi_{pc} = \text{LDQ } \mathbf{r}_d, n(\mathbf{r}_s) \text{ and } \boxed{\mathbf{rd}(\mathbf{r}_s \oplus n)} \\
(\rho[\mathbf{r_m} \leftarrow \mathbf{upd}(\mathbf{r_m}, \mathbf{r}_d \oplus n, \mathbf{r}_s)], pc+1), & \text{if } \Pi_{pc} = \text{STQ } \mathbf{r}_s, n(\mathbf{r}_d) \text{ and } \boxed{\mathbf{wr}(\mathbf{r}_d \oplus n)} \\
(\rho, pc+n+1), & \text{if } \Pi_{pc} = \text{BEQ } \mathbf{r}_s, n \text{ and } \mathbf{r}_s = 0 \\
(\rho, pc+1), & \text{if } \Pi_{pc} = \text{BEQ } \mathbf{r}_s, n \text{ and } \mathbf{r}_s \neq 0
\end{cases}
$$

rd(*a*) true iff *a* aligned on an 8-byte boundary

Certifying the Safety of Programs

$$(\rho, pc) \rightarrow \begin{cases} (\rho[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op], pc+1), & \textit{if} \;\; \Pi_{pc} = \text{ADDQ} \;\; \mathbf{r}_s, op, \mathbf{r}_d \\ (\rho[\mathbf{r}_d \leftarrow \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)], pc+1), & \textit{if} \;\; \Pi_{pc} = \text{LDQ} \;\; \mathbf{r}_d, n(\mathbf{r}_s) \;\; \textit{and} \;\; \boxed{\mathbf{rd}(\mathbf{r}_s \oplus n)} \\ (\rho[\mathbf{r_m} \leftarrow \mathbf{upd}(\mathbf{r_m}, \mathbf{r}_d \oplus n, \mathbf{r}_s)], pc+1), & \textit{if} \;\; \Pi_{pc} = \text{STQ} \;\; \mathbf{r}_s, n(\mathbf{r}_d) \;\; \textit{and} \;\; \boxed{\mathbf{wr}(\mathbf{r}_d \oplus n)} \\ (\rho, pc+n+1), & \textit{if} \;\; \Pi_{pc} = \text{BEQ} \;\; \mathbf{r}_s, n \;\; \textit{and} \;\; \mathbf{r}_s = 0 \\ (\rho, pc+1), & \textit{if} \;\; \Pi_{pc} = \text{BEQ} \;\; \mathbf{r}_s, n \;\; \textit{and} \;\; \mathbf{r}_s \neq 0 \end{cases}$$

Figure 3: The Abstract Machine.

$$VC_{pc} = \begin{cases} VC_{pc+1}[\mathbf{r}_d \leftarrow \mathbf{r}_s \oplus op], & \textit{if} \;\; \Pi_{pc} = \text{ADDQ} \;\; \mathbf{r}_s, op, \mathbf{r}_d \\ \mathbf{rd}(\mathbf{r}_s \oplus n) \wedge VC_{pc+1}[\mathbf{r}_d \leftarrow \mathbf{sel}(\mathbf{r_m}, \mathbf{r}_s \oplus n)], & \textit{if} \;\; \Pi_{pc} = \text{LDQ} \;\; \mathbf{r}_d, n(\mathbf{r}_s) \\ \mathbf{wr}(\mathbf{r}_d \oplus n) \wedge VC_{pc+1}[\mathbf{r_m} \leftarrow \mathbf{upd}(\mathbf{r_m}, \mathbf{r}_d \oplus n, \mathbf{r}_s)], & \textit{if} \;\; \Pi_{pc} = \text{STQ} \;\; \mathbf{r}_s, n(\mathbf{r}_d) \\ (\mathbf{r}_s = 0 \Rightarrow VC_{pc+n+1}) \wedge (\mathbf{r}_s \neq 0 \Rightarrow VC_{pc+1}), & \textit{if} \;\; \Pi_{pc} = \text{BEQ} \;\; \mathbf{r}_s, n \\ Post, & \textit{if} \;\; \Pi_{pc} = \text{RET} \end{cases}$$

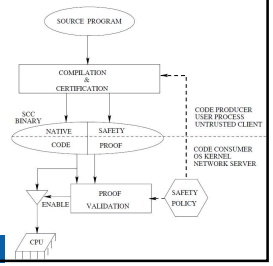
Figure 4: The Verification-Condition Generator.

Safety predicate $(\Pi, Pre, Post) = \forall r_0 \ldots \forall r_{10} \forall r_m Pre \Rightarrow VC_0$
Executing Π from any initial state satisfying *Pre* passes all checks & yields *Post*

Validating the Safety Proof

- **Use PCC system's simple theorem prover to generate proof of safety predicate**
  - Use Edinburgh Logical Framework (LF), a simple typed lambda calculus
- **Kernel computes safety predicate from the assembly code using the VC rules, then checks that the safety proof is valid proof of safety predicate**
  - Proof validation = typechecking

Ensures safety even if assembly code or the proof is tampered with



Application: Network Packet Filters

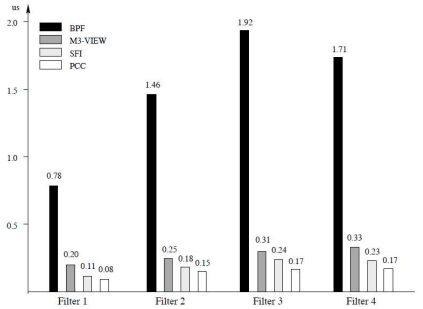
Safety Policy

1. Memory reads restricted to packet & statically-allocated scratch memory
2. Memory writes limited to scratch memory
3. All branches are forward
4. Reserved & callee-saved registers are not modified

Precondition

1. Packet length is ≥ 64 bytes and $< 2^{32}$ bytes
2. OK to read each byte in packet
3. OK to write each byte in 16-byte (aligned) scratch memory
4. No aliasing between packet and scratch memory

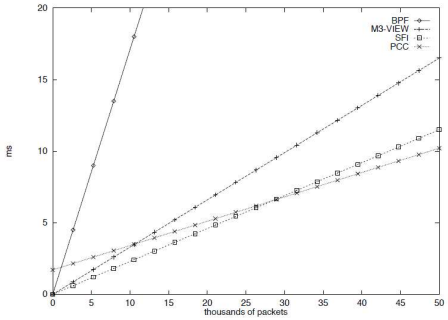
Average Per-Packet Run Time



- BPF: BSD Packet Filter, encode filter in Domain-specific language & run in BPF interpreter that does static checks
- M3-VIEW: Use type-safe Modula-3, some run-time checks
- SFI: Software Fault Isolation, run-time mem safety checks

Startup Cost Amortization

Packet Filter		1	2	3	4
Instructions		8	15	47	28
Binary Size (bytes)		385	516	1024	814
Validation Cost	Time (μs)	780	1070	2350	1710
	Space (KB)	5.5	8.7	24.6	15.1



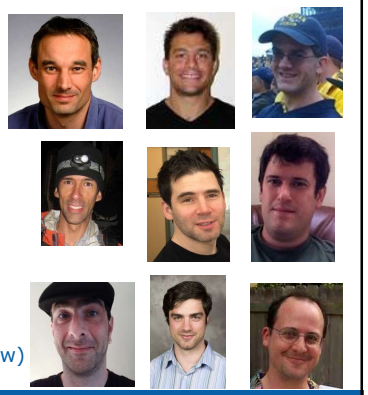
Discussion

- **Can extend to codes with loops**
  - via a table that maps each backward-branch to a loop invariant that is part of proof
- **Proof size is a concern**
- **PCC advantages**
  - Eliminates run-time checks
  - Tamper-proof
  - Safety for code written in any language
  - Safety properties beyond memory protection

Final Thought: "Ideas from PL are destined to become increasingly critical for robust & good-performing systems"

Application Performance and Flexibility on Exokernel Systems
[SOSP'97]

- **Frans Kaashoek** (MIT)
- **Dawson Engler** (Stanford)
- **Greg Ganger** (CMU)
- **Hector Briceno** (Visioglobe)
- **Russell Hunt** (??)
- **David Mazieres** (Stanford)
- **Thomas Pinckney** (eBay)
- **Robert Grimm** (NYU)
- **John Jannotti** (Brown)
- **Kenneth Mackenzie** (D.E. Shaw)



Exokernel Architecture [SOSP'95]

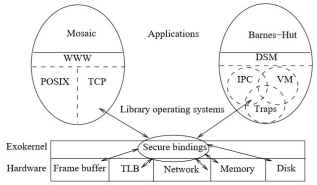


Figure 1: An example exokernel-based system consisting of a thin exokernel veneer that exports resources to library operating systems through secure bindings. Each library operating system implements its own system objects and policies. Applications link against standard libraries (*e.g.*, WWW, POSIX, and TCP libraries for Web applications) or against specialized libraries (*e.g.*, a distributed shared memory library for parallel applications).

- **Exokernel protects resources**
- **Unprivileged libraries (libOSes) provide traditional OS abstractions---apps select only what they need, can tailor**

Exokernel Principles

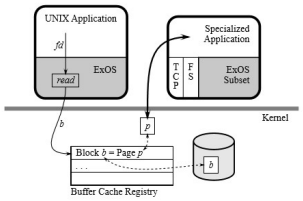
- **Separate protection from management**
  - Primitives at level of HW (e.g., disk blocks)
  - Resource management only for protection: allocation, revocation, sharing, ownership tracking
- **Expose allocation to applications**
- **Expose (physical) names to applications**
- **Expose revocation**
  - Apps choose which instance of resource to give up
- **Expose system info apps can't easily derive locally**
  - E.g., which pages cache file blocks, LRU info

Questions Addressed by Paper

- **Can ambitious apps get significant performance gains?**
- **Do traditional apps suffer reduced performance?**
- **Does lack of centralized management policy for shared OS structures lower the integrity of the system?**

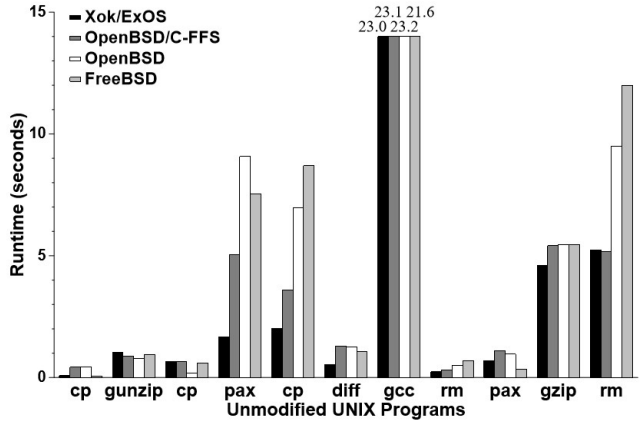
Multiplexing Stable Storage

- **Safely multiplex disks among multiple library file systems**
- **XN provides access at level of disk blocks, exporting a buffer cache registry**
  - Challenge: Application-defined metadata layouts
  - Untrusted Deterministic Functions (UDFs) translate metadata into set of blocks
  - Registry enables coherent caching of disk blocks across apps

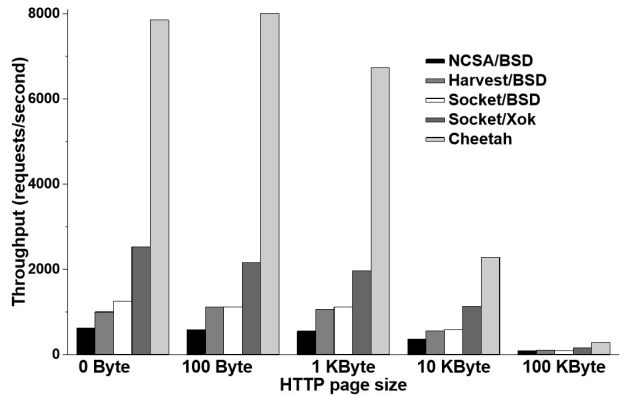


"...still exhibited what we have come to appreciate as an indication that applications do not have enough control: the system made too many tradeoffs."

Performance of Unmodified UNIX Apps



Cheetah HTTP/1.0 Server



Global Performance

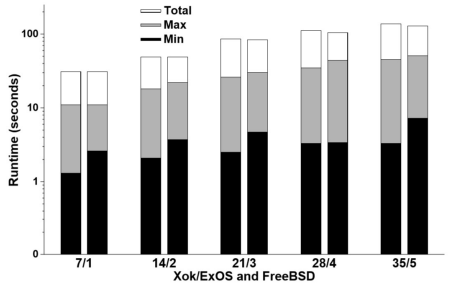


Figure 4: Measured global performance of Xok/ExOS (the first bar) and FreeBSD (the second bar), using the first application pool. Times are in seconds and on a log scale. *number/number* refers to the the total number of applications run by the script and the maximum number of jobs run concurrently. **Total** is the total running time of each experiment, **Max** is the longest runtime of any process in a given run (giving the worst latency). **Min** is the minimum.

Friday's Paper

The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

Austin Clements, Frans Kaashoek, Nickolai Zeldovich, Robert Morris, and Eddie Kohler

SOSP'13 Best Paper