Exokernel and Safe Kernel Extensions

Phil Gibbons
15-712 F15
Lecture 17

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

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

“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

\[
\begin{align*}
op & ::= \text{op} & i \in \{0 \ldots 10\} & \text{ADDQ, SUBQ, AND, OR, SLL, SRL} \\
br & ::= \text{br} & & \text{BEQ, BNE, BGE, BLT} \\
\text{instr} & ::= \text{instr} & & \text{LDQ, STQ, n(r_d), al, r_v, op, r_d, br, r_v, n, RET} \\
\end{align*}
\]

\[(p,pc) \rightarrow \begin{cases} 
(p[x_d \leftarrow r_x \oplus \text{op}], pc + 1), & \text{if } \Pi_{pc} = \text{ADDQ } r_x, \text{op}, r_d \\
(p[x_d \leftarrow \text{ADDQ}(r_x, pc + 1)], \text{if } \Pi_{pc} = \text{LDQ } r_x, n(r_d) \text{ and } r_d(r_x)) \\
(p[x_d \leftarrow \text{STQ}(r_x, n(r_d))], \text{if } \Pi_{pc} = \text{STQ } r_x, n(r_d) \text{ and } r_d(r_x)) \\
(pc + 1), & \text{if } \Pi_{pc} = \text{BEQ } r_x, n \text{ and } r_x = 0 \\
(pc + 1), & \text{if } \Pi_{pc} = \text{BEQ } r_x, n \text{ and } r_x \neq 0 \\
\end{cases}\]

\(rd(a) \text{ true iff a aligned on an 8-byte boundary}\)
Certifying the Safety of Programs

\[
\begin{align*}
(p, d) \rightarrow & \quad \text{if } \Pi_{\text{Pre}} = \text{ADDQ } r_{x}, q_{y}, r_{z} \\
(p, d) \rightarrow & \quad \text{if } \Pi_{\text{Pre}} = \text{LDQ } r_{x}, n(r_{i}) \\
(p, d) \rightarrow & \quad \text{if } \Pi_{\text{Pre}} = \text{STQ } r_{x}, n(r_{i}) \\
(p, d) \rightarrow & \quad \text{if } \Pi_{\text{Pre}} = \text{BEQ } r_{x}, n \\
(p, d) \rightarrow & \quad \text{if } \Pi_{\text{Pre}} = \text{RET }
\end{align*}
\]

Figure 3: The Abstract Machine.

\[\forall r_{0} \ldots \forall r_{19} \forall r_{m} \text{Pre } \Rightarrow \text{VC}_{0}\]

Executing \Pi 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

<table>
<thead>
<tr>
<th>Packet Filter</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>8</td>
<td>15</td>
<td>47</td>
<td>28</td>
</tr>
<tr>
<td>Binary Size (bytes)</td>
<td>385</td>
<td>516</td>
<td>1024</td>
<td>314</td>
</tr>
<tr>
<td>Validation Time (μs)</td>
<td>1780</td>
<td>1070</td>
<td>2250</td>
<td>1710</td>
</tr>
<tr>
<td>Cost Space (KB)</td>
<td>3.5</td>
<td>8.7</td>
<td>24.6</td>
<td>15.1</td>
</tr>
</tbody>
</table>

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 (Visiglobe)
- Russell Hunt (??)
- David Mazieres (Stanford)
- Thomas Pinckney (eBay)
- Robert Grimm (NYU)
- John Jannotti (Brown)
- Kenneth Mackenzie (D.E. Shaw)

Exokernel Architecture [SOSP’95]

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

Figure 1: An example exokernel-based system consisting of a thin exokernel kernel that exports resources to library operating systems through access 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 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

- Xok/ExOS
- OpenBSD/IC-FFS
- OpenBSD
- FreeBSD

<table>
<thead>
<tr>
<th>Program</th>
<th>Xok/ExOS</th>
<th>OpenBSD/IC-FFS</th>
<th>OpenBSD</th>
<th>FreeBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp</td>
<td>21.1</td>
<td>23.1</td>
<td>21.6</td>
<td>23.2</td>
</tr>
<tr>
<td>gunzip</td>
<td>23.0</td>
<td>23.0</td>
<td>23.2</td>
<td>23.2</td>
</tr>
</tbody>
</table>

Cheetah HTTP/1.0 Server

Throughput (requests/second)

- NCSA/bsd
- Harvest/bsd
- Socket/bsd
- Socket/Xok
- Cheetah

Global Performance

- Total
- Max
- Min

<table>
<thead>
<tr>
<th>Process</th>
<th>7/1</th>
<th>14/2</th>
<th>21/3</th>
<th>28/4</th>
<th>35/5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xok/ExOS and FreeBSD</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

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 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