Motivation

• Applications often vulnerable to security exploits

• Solution: restrict application access to the minimum amount of data needed
  • Least privilege principle
In a nutshell

- Fine-grained *virtualization* layer confines security threats
  - All executed code is verified
  - Additional security guards are added to the runtime image
  - All system calls are verified according to a tight policy
Outline

• Introduction
• Security architecture
  • Security through virtualization
  • Software-based fault isolation (SFI)
  • System call interposition
• Evaluation
• Related work
• Conclusion
Introduction

- Software security is a challenging problem
  - Many different forms of attacks exist
  - Low-level bugs are omni-present
  - Current security practice is reactive
- We present a pro-active approach to security
  - Catch exploits before they can cause any harm
Protection through virtualization

- Virtualization confines and secures applications
- Use a user-space virtualization system
  - Secure all code and authorize all system calls
Security Architecture

- Layered security concept
  - User-space software-based fault isolation
  - System call interposition framework
  - System call authorization
Software-based fault isolation

- SFI implemented as a user-space library
- All code is translated before it is executed
  - Code is checked and verified on the fly
  - All unsafe instructions are encapsulated or rewritten
    - Check targets and origins of control flow transfers
    - Illegal instructions halt the program
SFI: Additional guards

- Translator adds guards that protect from malicious attacks against the SFI platform and enhance security guarantees
  - Secure control flow transfers
  - Signal handling
  - Executable bit removal
  - Address space layout randomization
  - Protecting internal data structures
SFI: Control transfers

- Verify return addresses on stack
  - Use a shadow stack to store original/translated addresses
  - Protects from Return Oriented Programming

- Secure control flow transfers
  - Check target and source locations for valid transfer points
  - Protects from code injection through heap-based/stack-based overflows
SFI: Signal handling

- Catch signals and exceptions
  - Redirect to installed handlers if signal is valid
  - Protects from break-outs out of the sandbox
SFI: Executable bit removal

• Executable bit removed for libraries and application
  • Only libdetox and code-cache contains executable code

• Part of the protection against code-injection
SFI: ASLR

• Address space layout randomization randomizes the runtime memory image
  • Probabilistic measure that makes attack harder
SFI: Internal data structures

• All internal data structures are protected
  • Context transfer to (translated) application code protects all internal data structures
  • Write permissions to all internal memory is removed

• Protects from code-injection and attacks against the virtualization platform
SFI: Added protection

- These additional guards protect from
  - Code injection (stack-based / heap-based)
  - Return-oriented programming
  - Execution of illegal code
  - Attacks against the virtualization platform
System call interposition

- Implemented on top of SFI platform

- All system calls & parameters are checked
  - Dangerous system calls are redirected to a special implementation inside the virtualization library

- System call authorization
  - System calls are authorized based on a user-definable per-process policy

- Protects from data attacks
Outline

• Introduction
• Security architecture
  • Security through virtualization
  • Software-based fault isolation (SFI)
  • System call interposition
• Evaluation
• Related work
• Conclusion
libdetox

• Approach implemented as a prototype
• Built on top of fastBT system
  • Additional security hardening
  • Guards implemented in the translation process
  • Dynamic guards extend the dynamic control flow transfer logic
Evaluation

- SPEC CPU2006 benchmarks used to evaluate overheads
- Apache plus policy used to evaluate server performance
- All benchmarks were executed on Ubuntu 9.04 on an E6850 Intel Core2Duo CPU @ 3.00GHz, 2GB RAM and GCC version 4.3.3
SPEC CPU2006

- Benchmarks executed with well-defined policy

- Three configurations:
  - Binary translation (BT) only
    - no security extensions
    - shows cost of translation & control flow transfers
  - libdetox
    - standard security features
  - libdetox + internal memory protection
    - securing internal data structures
    - all transfers from the application code to the libdetox code are protected
## SPEC CPU2006

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BT</th>
<th>libdetox</th>
<th>+ mprot</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>55.97%</td>
<td>59.88%</td>
<td>74.69%</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>3.89%</td>
<td>5.39%</td>
<td>5.54%</td>
</tr>
<tr>
<td>429.mcf</td>
<td>-0.49%</td>
<td>0.49%</td>
<td>0.25%</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>6.17%</td>
<td>9.20%</td>
<td>9.20%</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>23.72%</td>
<td>27.22%</td>
<td>31.27%</td>
</tr>
<tr>
<td>454.calculix</td>
<td>-1.68%</td>
<td>-0.56%</td>
<td>-1.12%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>6.00%</strong></td>
<td><strong>6.39%</strong></td>
<td><strong>8.21%</strong></td>
</tr>
</tbody>
</table>

- Average overhead is low
- Most overhead comes from the BT
- Even worst-case behavior (perlbench) is manageable

* Average is calculated over all 28 SPEC CPU2006 benchmarks
Apache2

- Fully protected Apache 2.2.11 is evaluated using the ab benchmark
  - Each file is received 1'000'000 times
    - test.html (static, 1.7kB)
    - phpinfo.php (small, dynamic PHP file)
    - picture.png (static, 242 kB)
### Apache2

- Low overhead for real-world server application
- Throughput highly depends on payload
  - Both for virtualized and native executions
Related Work

- **Full system translation (VMWare, QEMU, Xen)**
  - Virtualizes a complete system, management overhead, data sharing problem

- **System call interposition (Janus, AppArmor)**
  - Only system calls checked, code is unchecked

- **Software-based fault isolation (Vx32, Strata)**
  - Only a sandbox is not enough, additional guards and system call authorization needed

- **Static binary translation (Google's NaCL)**
  - Limits the ISA, special compilers needed
Conclusions

- Combining SFI and policy-based system call authorization builds low overhead virtualization platform
  - Virtualization based on programs, not systems
  - System image is shared with a single configuration

- Fine-grained access control to data / properties

- Opens door to new approaches of security
  - Highly customizable and dynamic
Questions

• Libdetox as an implementation prototype supports full IA-32 ISA without kernel module
  • Source: http://nebelwelt.net/projects/libdetox/
Policy

System call definition:

5: open(string, int)
6: close(int)

open:

("/etc/apache2/\*", \*): allow
("/var/www/\*", \*): allow
("\*", \*): deny

close:

\*): allow
Policy: nmap

mode: whitelist /* not listed: abort program */
brk(*): allow /* memory management */
mmap2(*, *): allow
munmap(*, *): allow
close(*): allow
ioctl(*, TIOCGPGRP, *): allow
open("/dev/tty", *): allow
open("/etc/host.conf", *): allow
open("/etc/hosts", *): allow
open("/usr/share/nmap/nmap-services", *): allow
...
read(*, *): allow
stat64("/etc/resolv.conf", *): allow
stat64("/home/test/.nmap/nmap-services", *): allow
...
write(*, *): allow
socketcall(PF NETLINK, SOCK RAW, 0): allow /* net */
socketcall(PF INET, SOCK STREAM, IPPROTO TCP): allow
socketcall(PF FILE, SOCK STREAM | SOCK CLOEXEC | SOCK NONBLOCK, 0): allow
...
SFI in a nutshell

- Translates individual basic blocks
- Verifies code source / destination
- Checks branch targets and origins

Original code

Translator

Mapping table

Indirect control flow transfers use a dynamic check to verify target and origin

Code cache