Protection via Separation

Lujo Bauer
18-732
Spring 2015
When There Is No Protection…
Separation
Separation
Resources That Need Protection

- Multiprogramming increased the need for protection
- Resources that need protection
  - Memory
  - Shared IO devices
    - Disks (simultaneously sharable)
    - Printers (serially sharable)
  - Shared programs and modules
  - Networks
  - Shared data
Methods of Separation

- **Physical**
  - Multiple printers, disks, ...

- **Temporal**
  - Insecure before 12pm; secure after 12pm

- **Logical**
  - OS/environment provides isolation

- **Cryptographic**
  - Data and computation concealed cryptographically
Degrees of Separation

- **None**
  - All resources are fully shared

- **Complete isolation**
  - Programs are completely oblivious to each other

- **Binary sharing**
  - Resources are either public or private

- **Limited sharing**
  - Fine-grained control
  - Authorization checked on a per-access basis
  - E.g., ACLs, capabilities

- **Usage control**
  - Control not just access to resources, but how they’re used
Granularity of Separation

- What are the objects that are being separated from each other?
  - Programs
  - Sets of programs
  - Parts of a program
  - OSes
Mechanisms for Separation

- **Hardware**
  - Memory protection
- **Software**
  - Sandboxing
- **Hardware + software**
  - Virtual machines

**Key component:** *Reference monitor*
- Mediates requests across separation boundaries
- Enforces separation/confinement

**BIG design space. We will explore various points.**

**Today**

**Next couple of lectures**
Memory Protection

Logical separation, between programs, using hardware (and software)
Memory Protection – Fence

- Boundary between user programs and operating system

- Fence fixed for all programs and OSes
Memory Protection – Fence

- A slight refinement

- One-way protection
- No protection within user program space
Memory Protection – Base/Bounds Registers

- If one fence is good, two are better

- Programs protected from each other
- Context switch exchanges registers
- Programs can still hurt themselves
Memory Protection – Base/Bounds Registers

- If two are good, four are better

<table>
<thead>
<tr>
<th>Memory</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1 data</td>
<td>Program 2 data</td>
</tr>
<tr>
<td>Program 1 code</td>
<td>Program 2 code</td>
</tr>
</tbody>
</table>

- Data base register
- Data bounds register
- Code base register
- Code bounds register
Memory Protection – Base/Bounds Registers

- If four are good, … ?

- E.g., separate data into read-only and read-write
  - Instructions have to include pointer to data space
Memory Protection – Tagging

- If four are good, ... ?

- E.g., separate data into read-only and read-write
  - Instructions have to include pointer to data space

- In the limit, each word of memory can be in own domain
  - E.g., labeled as read-only, read-write, or execute
Memory Protection – Segmentation

- Program divided into many logical, named pieces
- Addresses have the form `<name, offset>`
- OS maintains translation table
Memory Protection – Segmentation

- Program divided into many logical, named pieces
- Addresses have the form `<name, offset>`
- OS maintains translation table

**Advantages**
- All references by name – OS mediates each access
- Segments can be moved around or swapped to disk
- Segments can be shared
Memory Protection – Segmentation

- Program divided into many logical, named pieces
- Addresses have the form `<name, offset>`
- OS maintains translation table

Disadvantages
- Fragmentation
- Inefficient bounds checking
  - Offset may be beyond end of segment
Memory Protection – Paging

- Program divided into equal-sized *pages*
- Addresses have the form `<page, offset>`

- Fragmentation isn’t a problem
  - All pages (segments) have the same size

- **Addressing ensures offset is within page**
  - E.g., 1024-byte pages use 10 bytes for the offset

- **Logical segmentation isn’t preserved**
  - Pages don’t correspond to logical program segments
Memory Protection – Segmentation & Paging

- Combine efficiency of paging with protection of segmentation
- First divide into segments, then each segment into pages

- Intel x86 architecture implements paged segmentation …
- … but not widely used
Isolation Mechanisms

- **Hardware**
  - Memory Protection (virtual address translation, x86 segmentation)

- **Software**
  - Sandboxing
  - Language-Based

- **Hardware + Software**
  - Virtual machines

Software Fault Isolation
≈ memory protection in software
Software Fault Isolation (SFI) Goals

- **Confine faults inside distrusted extensions**
  - codec shouldn’t compromise media player
  - device driver shouldn’t compromise kernel
  - plugin shouldn’t compromise web browser

- **Allow for efficient cross-domain calls**
  - numerous calls between media player and codec
  - numerous calls between device driver and kernel

[Wahbe, Lucco, Anderson, Graham 1993]
SFI: Main Idea

• Allocate each untrusted program to a fault domain (contiguous memory space + unique identifier)

• Modify untrusted program to prevent it from writing or jumping to an address outside its fault domain
SFI: In More Detail

- Code divided into code and data segments
  - Jump targets restricted to code segment
  - Data addresses restricted to data segment

- Code and data segment addresses stored in dedicated registers

- Before executing any instruction that references memory
  - Compare segment of memory reference to stored segment addresses
  - Stop execution if addresses aren’t equal

- Applications
  - User-level file systems, Google’s NativeClient
Main Idea

Process Address Space

Module 2
Fault Domain 2

Module 1
Fault Domain 1

segment with id 2,
e.g., with top bits 010

segment with id 1,
e.g., with top bits 011
SFI Scheme: Segment Matching

- Check every mem access for matching seg ID
- Assume dedicated register dr, sr
  - not available to the program

Process Address Space

Module 2

Module 1

precondition:
segment-reg (sr) holds segment id 2

dr = addr
scratch = (dr >> 29)
compare scratch, sr
trap if not equal
dst = [dr]
SFI Scheme: Sandboxing

- Force top bits to match seg ID and continue
- No comparison is made

precondition:
\( sr \) holds segment id 2

\[
\begin{align*}
\text{dr} &= (\text{addr} \& \text{mask}) \\
\text{dr} &= (\text{dr} | \text{sr}) \\
\text{dst} &= [\text{dr}]
\end{align*}
\]
Soundness

- Segment matching/sandboxing code must always be run
- Dedicated registers must not be writeable by module
Segment Matching vs. Sandboxing

Segment Matching
- more instructions
- can pinpoint exact point of fault where segment ID doesn’t match

Sandboxing
- fewer instructions
- just ensures memory access stays in region (crash is ok)
## Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DEC-MIPS</th>
<th>DEC-ALPHA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fault Isolation Overhead</td>
<td>Protection Overhead</td>
</tr>
<tr>
<td>052.alvinn</td>
<td>FP 1.4%</td>
<td>33.4%</td>
</tr>
<tr>
<td>bps</td>
<td>FP 5.6%</td>
<td>15.5%</td>
</tr>
<tr>
<td>cholesky</td>
<td>FP 0.0%</td>
<td>22.7%</td>
</tr>
<tr>
<td>026.compress</td>
<td>INT 3.3%</td>
<td>13.3%</td>
</tr>
<tr>
<td>056.ear</td>
<td>FP -1.2%</td>
<td>19.1%</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>INT 2.9%</td>
<td>34.4%</td>
</tr>
<tr>
<td>008.espresso</td>
<td>INT 12.4%</td>
<td>27.0%</td>
</tr>
<tr>
<td>001.gcc1.35</td>
<td>INT 3.1%</td>
<td>18.7%</td>
</tr>
<tr>
<td>022.li</td>
<td>INT 5.1%</td>
<td>23.4%</td>
</tr>
<tr>
<td>locus</td>
<td>INT 8.7%</td>
<td>30.4%</td>
</tr>
<tr>
<td>mp3d</td>
<td>FP 10.7%</td>
<td>10.7%</td>
</tr>
<tr>
<td>psgrind</td>
<td>INT 10.4%</td>
<td>19.5%</td>
</tr>
<tr>
<td>qcd</td>
<td>FP 0.5%</td>
<td>27.0%</td>
</tr>
<tr>
<td>072.sc</td>
<td>INT 5.6%</td>
<td>11.2%</td>
</tr>
<tr>
<td>tracker</td>
<td>INT -0.8%</td>
<td>10.5%</td>
</tr>
<tr>
<td>water</td>
<td>FP 0.7%</td>
<td>7.4%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>4.3%</strong></td>
<td><strong>21.8%</strong></td>
</tr>
</tbody>
</table>

- store and jump checked
- load, store and jump checked
Is it counter-intuitive?

- Slow down “common” case of intra-domain control transfer in order to speed up inter-domain transfer
  - Check every load, store, jump within a domain

- Faster in practice than hardware when inter-domain calls are frequent
  - Context switches are expensive
  - Each cross-module call requires a context switch
Questions

- Do these mechanisms provide control flow integrity?
  - CFI lecture (Feb 3)

- Do these mechanisms provide information flow control?
  - Android security (Jan 29)
  - Web security (Feb 12, 17)
Take Away: Reference Monitors

- **External reference monitor**
  - e.g., OS component with hardware-support enforces memory separation among untrusted programs

- **Inlined reference monitor**
  - e.g., SFI mechanism enforces memory protection by rewriting untrusted program to ensure it accesses memory within its own space

- **Reference monitors can enforce other interesting security properties**
  - More examples in the next three lectures (Jan 29, Feb 3, Feb 5)
  - A general characterization on Feb 10
Sources