Unsafe Code

- We often want to execute code that we don’t really trust:
  1. The Algovoista search engine allows users to upload code to a server where it is executed as part of a query.
  2. Some operating systems allow packet filters and other executable code to be downloaded into the kernel.
  3. Applets can be downloaded as part of Web pages.
- Software Fault Isolation is an efficient solution to the problem of protecting trusted code from malicious downloaded code.
- The optimal research path: Academic research ⇒ Start-up ⇒ Sell out to Microsoft.

Attacks by Untrusted Code

We want to stop untrusted code from
1. writing data not its own,
2. reading data not its own (sometimes),
3. jumping to non-entry points,
4. making dangerous system calls.

Slide 12–1

Slide 12–2

Slide 12–3
Anonymous RPC Fault Isolation

- Modules are placed at random VM addresses in the same address space. If you can't find a module, you can't compromise it.
- Calls go through a jump-table to preserve anonymity.
- SEGV handling is slowed down to prevent brute-force attacks.

Slide 12–4

Language-Based Fault Isolation

- Castrated virtual machine + strong typing + verification assures that illegal pointers can't be manufactured and all branches go to the beginning of statements.
- Trusted and untrusted modules reside together in the same address space.
- Communication between address spaces is by means of fast (inter-address space) RPC.
- In the untrusted code, extra instructions are inserted before every indirect store and jump to make sure it won't write into other modules or jump to other modules.
- Various optimizations are used to reduce the overhead of checking instructions.
- System calls are patched to go through arbitration code to check that the calls are safe.

Slide 12–7
An efficient method for implementing a safe virtual machine in software, that embodies a general purpose memory protection model. The present invention runs on any general purpose computer architecture and will run an executable that has been developed for the virtual machine. The present invention compiles the executable into the native instructions of the hardware. During the compilation, specialized code sequences are added to the code using a technique called software fault isolation. A set of allowed behaviors and a set of responses to the undesirable actions will be created and written to memory. A series of optimizations are applied so that the translated code executes at nearly the native speed of the architecture, but the fault isolation sequences prevent it from engaging in undesirable actions. In particular, the memory protection model requires hardware support to enforce efficiently. Abstract: US Patent 5764477.

**SFI Overview**

- Compile to OmniVM code
- Compile to native code
- Insert SFI Check instrs.
- Check-code optimization
- Local/global optimization
- Execute on native hardware

- Slow, safe, native code
- Fast, safe, native code
- Faster, safe, native code

**Fault Domains**

- Each untrusted module resides in a fault domain. Each domain has a data segment and a code segment.
- All virtual addresses in a segment have the same upper k bits, the segment identifier.
- The code is modified so that it can only jump to targets in the code segment, and access only data in the data segment.
- I.e. all addresses in a module have the same segment identifier.
Fault Isolation I

- We must make sure that
  1. no store affects data in another segment,
  2. no jump target is outside the current segment.
⇒ All touched addresses must have the same segment identifier as the current module.
- No problem:
  - stores to static variables,
  - pc-relative branches.
- Problems:
  - stores through a register:
    \texttt{st } [r1+8], r2.
  - jumps through a register (returns).

Slide 12–12

Fault Isolation II

\begin{align*}
\text{st} & \quad \text{global, r1} \\
\text{br} & \quad \text{L1} \\
\text{st} & \quad [r2+8], r1 \\
\text{jmp} & \quad [r1+16] \\
\downarrow & \quad \\
\text{st} & \quad \text{global, r1} \\
\text{br} & \quad \text{L1} \\
& \quad \text{if } (\text{segid(r2+8)} \neq 567) \\
& \quad \text{halt} \\
\text{st} & \quad [r2+8], r1 \\
& \quad \text{if } (\text{segid(r1+16)} \neq 567) \\
& \quad \text{halt} \\
\text{jmp} & \quad [r1+16]
\end{align*}

- A trivial implementation of fault isolation.

Slide 12–13

Fault Isolation III

- Several algorithms for efficient sandboxing are given.
- The algorithms use \textit{dedicated registers}, not touched by any user code. Works well for RISCs with many registers.
- \textit{Segment Matching} pinpoints the offending instruction. Good for development. 4 dedicated registers, 4 extra instructions.
- \textit{Sandboxing} requires 5 dedicated registers, 2 extra instructions. The use of dedicated registers ensures that, should the untrusted code jump into the middle of the fault-checking code, nothing bad happens.

Slide 12–14

<table>
<thead>
<tr>
<th>Dedicated registers for Segment Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAR</td>
</tr>
<tr>
<td>DSB</td>
</tr>
<tr>
<td>DHR</td>
</tr>
</tbody>
</table>

```
\texttt{mov} \quad \texttt{DAR}, r1 \\
\texttt{sr} \quad r8, \texttt{DAR}, \texttt{DHR} \\
\texttt{cmp} \quad r8, 567 \\
\texttt{beg} \quad \texttt{L1} \\
\texttt{trap} \quad \texttt{[DAR], r2} \\
\texttt{L1: st} \quad [r1], r2
```
Optimization I

- Register-plus-offset stores require 3 sandboxing instructions:

\[
\text{st } [r1+offs], r2 \\
\text{add DAR, } r1, \text{ offs } \quad \# \text{ Compute } \text{reg}+\text{offs}
\]

and DAR, r1, DMR  \# Sandbox, as before

or DAR, DAR, DSR

\[
\text{st } [\text{DAR}], r2
\]

- We can do away without the extra add by sandboxing only r1 and inserting guard pages:

\[
\text{Size is range of offsets}
\]

<table>
<thead>
<tr>
<th>Unmapped guard zone pages</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>st [reg+offs], r2</td>
</tr>
</tbody>
</table>

| Unmapped guard zone pages |

Optimization II

- Sandbox sp only when it's set, not when it's used.

\[
\text{st } [\text{sp+16}], r8
\]

\[
\text{add } \text{sp, sp}, 16 \quad \# \text{ Change sp}
\]

or DAR, DAR, DSR

\[
\text{st } [\text{sp}], r8
\]

\[
\text{Don't sandbox!}
\]

- Uses of sp are more frequent than changes.
Process Resources

- A system call in untrusted code is transformed to an RPC to arbitration code that rejects unsafe calls:

<table>
<thead>
<tr>
<th>Bad Code</th>
<th>Nice Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>close(f)</td>
<td>f = open(...)</td>
</tr>
<tr>
<td>delete(g)</td>
<td>g = open(...)</td>
</tr>
<tr>
<td></td>
<td>a = read(f)</td>
</tr>
</tbody>
</table>

Safe bad code

- Arbitration Code Domain

Cross Fault-Domain Calls

- Calls from trusted to untrusted domains must be efficient.
  Every call/return goes through a jump table (in read-only memory), then through a stub which copies arguments, saves registers, etc.

Optimization III

- Don't sandbox sp when it's set, if a load/store using sp follows before any branch.
  L1: add sp, sp, 16  # Change sp
      1d [sp+16], r8
      beq L1:
  ↓
  L1: add sp, sp, 16  # No sandboxing!
      1d [sp+16], r8  # Fault caught by guard zone.
      beq L1:

- Without the load, the branches could construct an illegal sp.
- Removes 75% of checks from debugging code, 40% from optimized code.

Optimization IV

- If a pointer is incremented by a small amount every time around a loop, we only need to sandbox it once.
### Evaluation II

<table>
<thead>
<tr>
<th></th>
<th>Mips</th>
<th>Sparc</th>
<th>PPC</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFI</td>
<td>SFI</td>
<td>SFI</td>
<td>SFI</td>
<td>SFI</td>
</tr>
<tr>
<td>cc -0 vs. omnicc -0</td>
<td>1.14</td>
<td>1.03</td>
<td>1.05</td>
<td>1.02</td>
</tr>
<tr>
<td>gcc -0 vs. omnicc -0</td>
<td>1.01</td>
<td>0.92</td>
<td>1.05</td>
<td>1.02</td>
</tr>
<tr>
<td>cc -0 vs. omnicc</td>
<td>1.17</td>
<td>1.04</td>
<td>1.21</td>
<td>1.15</td>
</tr>
</tbody>
</table>

### OmniVM

- The VM code files contain annotations which provide sufficient information to derive the CFG: procedure entry points, code/data segments, live registers, function signatures.
- Registers: 16 int, 16 float.
- On the x86, some registers are mapped to memory locations.
- Endian-neutral.
- 32-bit immediate offsets. (Helps with x86 code generation).
- General compare-and-branch instruction.

### Evaluation III

<table>
<thead>
<tr>
<th>Program</th>
<th>SFI</th>
<th>SFI+load protection</th>
<th>Reserved register</th>
<th>Dynamic instr. count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ear</td>
<td>-1.2%</td>
<td>19.1%</td>
<td>0.2%</td>
<td>12.4%</td>
</tr>
<tr>
<td>Average</td>
<td>4.3%</td>
<td>21.8%</td>
<td>0.4%</td>
<td>10.5%</td>
</tr>
</tbody>
</table>

- 5 Registers were used for sandboxing.
- Variations are due to instruction cache mapping conflicts (maybe). SFI changes the mapping of instructions to cache lines.
Verification

- When we download code that has had fault-checking code added, we need to verify the correctness of this code.
- These three invariants must hold:
  1. Every unsafe instruction must use a dedicated register.
  2. Every path that contains a statement which sets the value of a dedicated register must contain a check before that path reaches a use of the register.
  3. Every check contains a recognized sequence of instructions.

Check invariants: 1) examine register usage, 2) dataflow reaching analysis, 3) examine the check sequences.

Summary

- Language independent.
- Processor dependent.
- Makes the Web safe for unsafe languages like C and C++.
- Low overhead.
- Mutually distrustful program modules share the same address space.
- Works well for processors with large register files.
- RISC-like instruction set + block moves, etc.

References I


References II