Software-Based Fault Isolation

Jinseong Jeon
fault isolation?

- So, fault isolation is the act of separating something faulty?
fault detection & isolation

- “monitor a system, identify when a fault has occurred, and pinpoint the type of fault and its location” – Wikipedia

- Since one single operation costs a lot, warfare systems are mandatory to continue commands, detecting and isolating faulty units.

```
hostile signal

antenna

controller/
 test signal generator

a warfare system

PDW

Since one single operation costs a lot, warfare systems are mandatory to continue commands, detecting and isolating faulty units.
```
software-based fault isolation

- Software-based fault isolation is the act of separating something possibly faulty.
Efficient Software-Based Fault Isolation

R. Vahbe, S. Lucco, T. E. Anderson, and S. L. Graham
SOSP ’93
fault isolation?

- need to incorporate independently developed software modules
  - micro-kernel design
    - BSD network packet filter
    - application-specific virtual memory management
    - Active Messages
  - extensible software
    - MS object linking and embedding system
    - Quark Xpress desktop publishing system
  - high I/O processes
    - POSTGRES
- need to prevent faults in extension code from corrupting other codes or permanent data while cooperating
- Hence, fault isolation is an act of separating distrusted extensions.
hardware-based fault isolation

- place each software module in its own address space
- communicate through Remote Procedure Call (RPC)
  - trap into the OS kernel,
  - copying each argument from the caller to the callee,
  - saving and restoring registers,
  - switching hardware address space,
  - trap back to user level.
software-based fault isolation

- load extension codes and their data into their own *fault domain*
  - fault domain = code segment + data segment
- **enforce security policies** that
  - a distrusted module is prohibited from writing or jumping outside its fault domain.
  - i.e. those distrusted modules cannot modify/execute each other’s data/code.
  - the only way to do is to use explicit cross fault-domain communication.
possible questions

- how to enforce such security policies?
  - by binary rewriting

- what to rewrite, and how?
  - unsafe instruction
    - that cannot be statically verified to be within the correct segment
  - use dedicated registers
    - segment matching
    - address sandboxing

- how to share process resources and data?
  - trusted arbitration code
  - virtual address aliasing (or, shared segment matching)

- how to communicate with other fault domains?
  - explicit cross-fault-domain RPC interface
  - stub and jump table
segment matching

- fault domain
  - = code segment + data segment
  - shares a unique pattern of upper bits, "segment identifier"

- insert checking code before every unsafe instruction
  - indirect jumps or stores, i.e. via registers of which value is determined at runtime

- pseudo code
  
  ```
  dedicated-reg ← target address
  scratch-reg ← (dedicated-reg >> shift-reg)
  compare scratch-reg and segment-reg
  trap if not equal
  store/jump using dedicated-reg
  ```

```
<table>
<thead>
<tr>
<th></th>
<th>0x83800000</th>
<th>0x83800001</th>
<th>0x83900000</th>
<th>0x83900001</th>
</tr>
</thead>
<tbody>
<tr>
<td>code segment</td>
<td>...</td>
<td>0x83800000</td>
<td>...</td>
<td>0x83900001</td>
</tr>
<tr>
<td>data segment</td>
<td>0x83800000</td>
<td>...</td>
<td>0x83900000</td>
<td>...</td>
</tr>
</tbody>
</table>
```
address sandboxing

- instead of checking, just setting the upper bits to the correct segment identifier
- in the section “Ensure, don’t check” at the next paper,
  - check = segment matching
  - ensure = address sandboxing

pseudo code

```plaintext
dedicated-reg ← dedicated-reg | segment-reg[^2]
store/jump using dedicated-reg
```
sharing

- process resources
  - allocated on a per-address-space basis, e.g. file handles
  - making OS aware of fault domain – not portable
  - modifying distrusted modules’ accesses into RPC calls
  - allowing “the” trusted part to perform a direct system call and share the result
  - seemed similar to the way of sharing the same result of system calls in N-Var. paper?

- data
  - read-only
    - trivial because load instructions are not either checked or sandboxed
  - read-write
    - lazy pointer swizzling
      - alias shared region into each fault domain (via the same low order bits)
    - shared segment matching
      - a bitmap that holds a mapping from fault domains to accessible segments
trusted stubs to handle RPC
- for each pair of fault domains
- stub: copy arguments, re/store registers, switch the exe. stack, validate dedicated regs
  but! no traps or address space switching (thus, cheaper than HW RPC)

jump tables to transfer control
- consists of jump instructions of which target address is legal, outside the domain
optimizations

- **guard zone**
  - virtual memory pages adjacent to the segment
  - unmapped! i.e. trapped if accessed
  - store value, offset(reg)
  - sandboxing reg only, rather than reg+offset

- **stack pointer as a dedicated register**
  - # setting stack pointer < # using stack pointer to form address
  - once sandboxing the stack pointer whenever it is set, then no sandboxing is required for any other uses of this register.

- **avoid sandboxing the stack pointer when modified by a small constant**
  - unless it is used to transfer control

- **removing sandboxing sequences from loops**
  - sounds like ABCD: Eliminating Array-Bound Checks on Demand
verification

- divide the program into unsafe regions
  - starting with any modification of dedicated store/jump register
  - ending with one of the followings
    - next instruction is a store/jump to dedicated register
    - next instruction is guaranteed not to be executed
    - no more instructions in the segment

- for each unsafe regions,
  check whether dedicated registers are valid at region exit
  - sounds like computing reaching definition analysis in a classic data-flow analysis
  then, checking reaching definition of dedicated registers spans the above definition
## encapsulation overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DEC-MIPS</th>
<th>DEC-ALPHA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fault Isolation</td>
<td>Protection</td>
</tr>
<tr>
<td></td>
<td>Overhead</td>
<td>Overhead</td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td>Register</td>
</tr>
<tr>
<td></td>
<td>Register Overhead</td>
<td>Overhead</td>
</tr>
<tr>
<td></td>
<td>Overhead</td>
<td>Overhead</td>
</tr>
<tr>
<td>052.alvinn</td>
<td>FP 1.4%</td>
<td>33.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bps</td>
<td>FP 5.6%</td>
<td>15.5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cholesky</td>
<td>FP 0.0%</td>
<td>22.7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>026.compress</td>
<td>INT 3.3%</td>
<td>13.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>056.ear</td>
<td>FP -1.2%</td>
<td>19.1%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>023.eqntott</td>
<td>INT 2.9%</td>
<td>34.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>008.espresso</td>
<td>INT 12.4%</td>
<td>27.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001.gcc1.35</td>
<td>INT 3.1%</td>
<td>18.7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>022.li</td>
<td>INT 5.1%</td>
<td>23.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>locus</td>
<td>INT 8.7%</td>
<td>30.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mp3d</td>
<td>FP 10.7%</td>
<td>10.7%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>psgrind</td>
<td>INT 10.4%</td>
<td>19.5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qcd</td>
<td>FP 0.5%</td>
<td>27.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>072.sc</td>
<td>INT 5.8%</td>
<td>11.2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tracker</td>
<td>INT -0.8%</td>
<td>10.5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>water</td>
<td>FP 0.7%</td>
<td>7.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>4.3%</strong></td>
<td><strong>21.8%</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- about 5%
- fairly correct prediction

(# additional instruction - # saved floating point interlock cycles) / cycle-per-second

original-execution-time-seconds
when to use SFI

- \((1-r) \ t_c = h \ t_d\)
  - \(t_d\): % of time spent in distrusted code (1.00, 100%)
  - \(t_c\): % of time spent in crossing among fault domains
  - \(h\): overhead of encapsulation (0.043)
  - \(r\): ratio of time of SFI RPC to that of “competing” HW RPC
Evaluating SFI for a CISC Architecture

S. McCamant and G. Morrisett
USENIX-SS ’06
CISC architectures

- padding with no-ops to enforce alignment constraints (power of two)
  - because CISC architectures allow various instruction streams, which makes SFI harder

- call placed at the end of chunks
  - because the next addresses are targets of returns
  - they also have low 4 bits zero due to 16 bytes align

- put unsafe operation and its corresponding check together in a chunk
  - atomic, i.e. unsafe op. must be followed by check; no dedicated registers required
optimizations

- three introduced by R. Wahbe et al.

- one-instruction address operations
  - choose code and data region tags that have only a single bit difference
  - then, address need to be cleared only, without being set
  - e.g. code: 0x10000000, data: 0x20000000
  - and $0x20ffffff, %ebx

- efficient returns
  - modern x86 processors have a shadow stack where return addresses are cached.
  - (as long as a single thread is running)

before: popl %ebx
        and $0x10ffffff0, (%esp)
        and $0x10ffffff0, %ebx
        Jmp *%ebx

after: and $0x10ffffff0, (%esp)
verification

- security property to check:
  - a program never jumps outside code segment
  - or writes outside its data segment

- for each position in the rewritten instruction steam,
  conservatively collect all possible contents of the processor’s registers
  at anytime when execution might reach that point
  - meet-over-all-paths (MOP) data-flow analysis?
rewriter implementation

```assembly
push %ebp
mov %esp, %ebp
mov 8(%ebp), %edx
mov 48(%edx), %edx
lea 1(%eax), %ecx
lea 1(%eax), %ecx
lea 0(%esi), %esi
lea 0(%esi), %esi
lea 0(%edi), %edi
lea 0(%edi), %edi
lea 48(%edx), %ebx
lea 0(%esi), %esi
lea 0(%edi), %edi
lea 0(%esi), %esi
lea 0(%esi), %esi
and $0x20ffffff, %ebx
and $0x20ffffff, %ebp
and $0x10fffffff0, (%ebp)
ret
```

- **no op**
- **rsvd**
- **sandbox because %ebp changed**
- **sandbox before return**
- **no sandbox because %esp is safe**
verifier implementation

- merely a finite-sate machine (with only two states?)

<table>
<thead>
<tr>
<th>strengthen</th>
<th>weaken</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Diagram for strengthen" /></td>
<td><img src="#" alt="Diagram for weaken" /></td>
</tr>
</tbody>
</table>

```c
andl $0x10ffffff0, (%ebp)
```

```c
pop %ebp
```
performance
case study

- VX A, an archiving system where archives contain their own decompressor
- uses a virtualized execution environment VX32 to isolate decompressor code modules
- VX32 relies on hardware support for protecting against unsafe writes
  - less portable; not supported in the 64-bit mode, but work in 32-bit compatibility mode
formal analysis

- using ACL2, a theorem-proving system, prove the soundness of verifier

- ACL2, a restricted subset of Common Lisp

- proof is a simplified model of the verifier, along with a simulator for x86 instruction set

- proves that
  if the verifier approves the rewritten code,
  it will only execute safe instructions for all possible input states
references

- Efficient Software-based Fault Isolation, R. Wahbe et al., 1993

- Several presentations by
  - Maitree Kanungo,
  - J. Garrett Morris,
  - Christopher Head,
  - Kenneth Chiu,
  - Cole Trapnell,
  - and an anonymous presenter