CMPSC 497: Software Fault Isolation

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Motivation

• Process separation is great for security
  ‣ But inter-process communication is costly
• Can we establish logical “fault domains” within a process?
  ‣ Provides both separation between fault domains
  ‣ And fast communication
• Approach: software-based isolation within the same process
Approach

- Enforcement doesn’t require context switches in the kernel
- Lower performance overhead
- Environment independent---portable
- Policies can depend on application semantics

[Diagram showing Kernelized, Wrapper, and Modified program]

Integrate reference monitor into program code – and protect from untrusted program code
Software Fault Isolation (SFI)

• Use an inlined reference monitor to isolate components into “logical” address spaces in a process
  ‣ Conceptually: check each read, write, & jump to make sure it is within the component’s logical address space

• Originally proposed in 1993 for MIPS [Wahbe et al. SOSP 93]
  ‣ PittSField extended it to x86 [McCamant & Morrisett 06]
Fault Domains

• Each domain is a “logical” address space within a process’s address space
  ‣ Separate Code and Data Regions (Harvard architecture)
  ‣ Code region is readable and executable
    • Why the code region has to be unwritable?
  ‣ Data region is readable and writable
SFI Policy

Fault Domain

- **Code Region**: (readable, executable)
- **Data Region**: (readable, writable)

1. All jumps remain in CR
2. Reference monitor not bypassed by jumps

All R/W remain in DR [DB, DL]
void interp(int pc, reg[], mem[], code[]) {

    while (true) {
        if (pc < CB) exit(1);
        if (pc > CL) exit(1);
        int inst = code[pc], rd = RD(inst), rs1 = RS1(inst),
                   rs2 = RS2(inst), immed = IMMED(inst);
        switch (opcode(inst)) {
            case ADD: reg[rd] = reg[rs1] + reg[rs2]; break;
            case LD: int addr = reg[rs1] + immed;
                    if (addr < DB) exit(1);
                    if (addr > DL) exit(1);
                    reg[rd] = mem[addr];
                    break;
            case JMP: pc = reg[rd]; continue;
            ...
        }
        pc++;
    }
}
Interpretation

• Interpret programs written in a particular language
  ‣ Execution engine interprets each command, and checks that each operation is safe before doing it

• Examples
  ‣ SafeTcl, old Java implementations, Perl (sometimes)
  ‣ and a lot of scripting languages
  ‣ …
Pros & Cons of Interpreter

Pros:

- Easy to implement (small TCB)
- Works even with binaries (high-level language-independent)
- Easy to enforce other aspects of OS policy

Cons:

- Terrible execution overhead (x25? x70?)
- But it’s a start.
Partial Evaluation (PE)

• A technique for speeding up interpreters
  ‣ Specialize a program with respect to part of input that is **statically known**

• Example

```c
int f (int x, int i) {
    if (x>0) return i;
    else return (i+1);
}

... a = f(10, b) ...  same as a = b
... a = f(-10, c) ...  same as a = c + 1
```
Partial Evaluation for Faster SFI

• We know what the code is.
• Specialize the interpreter to the code.
  ‣ Unroll the loop – one copy for each instruction
  ‣ Specialize the switch to the instruction
  ‣ Compile the resulting code
Example PE

Original Binary:

0: add r1, r2, r3
1: ld r4, r3(12)
   ...

Interpreter:

while (true) {
    if (addr < DB) exit(1);
    if (addr > DL) exit(1);
    ...
}

Specialized interpreter:

addr = reg[3] + 12;
if (addr < DB) exit(1);
if (addr > DL) exit(1);
reg[4] = mem[addr];

Resulting Compiled Code:

add r1, r2, r3
add r5, r3, 12
cmp r5, DB
bj _exit
cmp r5, DL
ja _exit
ld r4, r5(0)
...
IRM via Program Rewriting

- The rewritten program should satisfy the desired security policy
- Examples:
  - Source-code level
    - CCured [Necula et al. 02]
  - Java bytecode-level rewriting: PoET [Erlingsson and Schneider 99]; Naccio [Evans and Twyman 99]
Enforcing SFI Policy

- Insert monitor code into the target program before unsafe instructions (reads, writes, jumps, ...)

- Code:
  
  ```
  [r3+12] := r4 //unsafe mem write
  
  r10 := r3 + 12
  if r10 < DB then goto error
  if r10 > DL then goto error
  [r10] := r4
  ```
SFI: Binary Rewriting

- A hand-written, specialized binary rewriter
  ‣ Insert monitor code into the target program before dangerous instructions

```
0: add r1,r2,r3
1: ld r4,r3(12)
...
```

```
add r1,r2,r3
add r5,r3,12
cmp r5,DB
jb _exit
cmp r5,DL
ja _exit
ld r4,r5(0)
...
```
Optimizations

• Naïve SFI is OK for security
  ‣ But the runtime overhead is too high

• Performance can be improved through a set of optimizations
Special Address Patterns

- Both code and data regions form contiguous segments
  - Upper bits are all the same and form a region ID
  - Address validity checking: only one check is necessary
- Example: DB = 0x12340000; DL = 0x1234FFFF
  - The region ID is 0x1234
  - “[r3+12]:= r4” becomes

```plaintext
r10 := r3 + 12
r11 := r10 >> 16 // right shift 16 bits to get the region ID
if r11 <> 0x1234 then goto error
[r10] := r4
```
Ensure, So No Check

- Force the upper bits in the address to be the region ID
  - Called **masking**
  - No branch penalty
- Example: \( DB = 0x12340000 \); \( DL = 0x1234FFFF \)
  - “\([r3+12]:= r4\)” becomes

\[
\begin{align*}
  r10 & := r3 + 12 \\
  r10 & := r10 \ & \ 0x0000FFFF \\
  r10 & := r10 \ | \ 0x12340000 \\
  \{ r10 \} & := r4
\end{align*}
\]

Force the address to be in data region
"Good" programs won’t get affected

- For bad programs, we do not care about whether its semantics are destroyed

- PittSField reported 12% performance gain for this optimization

- **Cons**: does not pinpoint the policy-violating instruction
One-Instruction Masking

• Idea
  ‣ Make the region ID to have only a single bit on
  ‣ Make the zero-tag region unmapped in the virtual address space

• Benefit: cut down one instruction for masking

• Example: $DB = 0x20000000$ ; $DL = 0x2000FFFF$
  ‣ Region ID is $0x2000$
  ‣ “[$r3+12] := r4” becomes

\[
\begin{align*}
  r10 & := r3 + 12 \\
  r10 & := r10 \& 0x2000FFFF \\
  [r10] & := r4
\end{align*}
\]

  ‣ Result is an address in DR or in the (unmapped) zero-tag region

• PittSField reported 10% performance gain for this optimization
Fault Isolation vs. Protection

• Protection is fail stop
  ‣ Sandbox reads, writes, and jumps
  ‣ Guarantee integrity and confidentiality
  ‣ 20% overhead on 1993 RISC machines
  ‣ XFI JPEG decoder: 70-80%

• Fault isolation: covers only writes and jumps
  ‣ Guarantee integrity, but not confidentiality
  ‣ 5% overhead on 1993 RISC machines
  ‣ XFI JPEG decoder: Writes only: 15-18%

• As a result, most SFI systems do not sandbox reads
Risk of Indirect Jumps

- **Worry**: what if the return address is modified so that the `ret` instruction jumps directly to the address of “r[10] := r4”?
  - The attack bypasses the masking before “r[10] := r4”!
  - If attacker can further control the value in r10, then he can write to arbitrary memory location
- In general, any computed jump might cause such a worry
  - `jmp %eax`
- BTW, direct jumps (pc-relative jumps) are easy to deal with – Why?

```
  r10 := r3 + 12
  r10 := r10 & 0x2000FFFF
  [r10] := r4
  ...
  ret
```
The Original SFI Solution

• Make r10 a dedicated register [Wahbe et al. 1993]
  ‣ r10 only used in the monitor code, not used by application code
  ‣ Also maintain the invariant that r10 always contains an address with the correct region ID before any computed jumps
    • So that even if a computed jump targets the middle of a pseudoinstruction, an address with the correct region ID will be used

• Cons?
  ‣ Reduce the number of registers available to application code
  ‣ OK for most RISC machines (E.g., MIPS has 32 registers)
  ‣ x86-32 has only 8 integer registers (6 general purpose ones);
    • x86-64: 16
A Solution for x86 (PittSField)

- Divide the code into chunks of some size
  - E.g., 16 bytes
- Make unsafe ops and their checks stay within one chunk
  - E.g., “r10 := r10 & 0x2000ffff; \([r10] := r4\)”
- Mask jump targets so that they are aligned: multiples of the chunk size
  - E.g., “jmp r5” becomes
    
    \[
    \begin{align*}
    r5 &:= r5 & 0x1000FFFF0 \\
    \text{jmp} &\ r5
    \end{align*}
    \]
    
    Note: the above assumes the region ID for the code region is 0x1000; a single instruction for sandboxing and alignment requirement

What about “ret”?
Downside of Alignment

• All legitimate jump targets have to be aligned
  ‣ No-op instructions have to be inserted sometimes
  ‣ For example: “i1; i2; i3”
    • Suppose both i1 and i3 are possible jump targets
    • Then it becomes “i1; i2; nop; nop; …; nop; i3”

• Cons: Slows down execution and increases code size
Jumping Outside of Domain

• Sometimes need to invoke code outside of the domain
  ‣ For system calls; for communication with other domains
  ‣ **Danger:** Cannot allow untrusted code to invoke code outside of the fault domain arbitrarily

• Idea:
  ‣ Insert a jump table into the (immutable) code region
  ‣ Each entry is a control transfer instruction whose target address is a legal entry point outside of the domain
A Fixed Jumptable (Trampoline)

- For example
  - Trampolines for system calls: fopen; fread; …
  - Trampolines for communication with other fault domains

Fault Domain

- Trampolines
- Code Region
- Data Region

stubs to trusted routines
Trusted Stubs

- Stubs are outside of the fault domain
  - Why?
- Stubs can implement security checks
  - E.g., can restrict fopen to open files only in a particular directory
  - Or can disallow fopen completely
    - Just not install a jump table entry for it
  - It can implement system call interposition
Trust in the SFI Rewriter?

- **SFI rewriter**: insert checks before unsafe instructions
  - It is in the TCB
- The rewriter can be moved out of the TCB
  - Construct a verifier that verifies the result of the rewriter
  - The verifier verifies that the necessary checks are there
- Q: Can we trust the verifier?
  - Prove its correctness using program verification
Google Native Client (NaCl)

- New SFI service in Chrome
  - [Yee et al. Oakland 09]
- Goal: Download native code and run it safely in the Chrome browser
  - Much safer than ActiveX controls
  - Much better performance than JavaScript, Java, etc.
NaCl: Code Verification

• Code is verified before running
  ‣ Allow restricted subset of x86 instructions
    • No unsafe instructions: memory-dependent jmp and call, privileged instructions, modifications of segment state …
  ‣ Ensure SFI checks are correctly implemented for memory safety
NaCl Sandboxing

• x86-32 sandboxing based on hardware segments
  ‣ Sandboxing reads and writes for free
  ‣ 5% overhead for SPEC2000

• However, hardware segments not available in x86-64 or ARM
  ‣ Still need masking instructions [Sehr et al. 10]
  ‣ x86-64/ARM: 20% for sandboxing memory writes and computed jumps
NaCl SDK

- Modified GCC tool-chain
  - Inserts appropriate masks, alignment requirements
- Trampolines allow restricted system-call interface and also interaction with the browser
  - Pepper API: access to the browser, DOM, 3D acceleration, etc.
Questions for SFI

• Binary rewriting on off-the-shelf binaries
  ‣ All current SFI implementations require the cooperation of the code producer

• What happens with discontiguous hunks of memory?

• Does this really scale to secure systems?
  ‣ So that we can partition a large system into domains of least privileges