Evaluating SFI for a CISC Architecture

Stephen McCamant and Greg Morrisett
smcc@csail.mit.edu, greg@eecs.harvard.edu
MIT CSAIL and Harvard EECS

Outline
SFI as a security technique
Classic (RISC) SFI
A CISC-compatible approach
PittSFIeld implementation
Machine-checked proof
Conclusion

Software security: isolation
How can I keep a piece of code from doing bad things?
Author might be malicious, or code might be subverted by malicious input
Identify legal interfaces; how to limit interaction to them?

Application: future-proof archives
Embed decompressor in .zip file so it’s always available [Ford, 2005]
How to safely execute untrusted library?

Well-known isolation techniques
OS process abstraction
  + Robust hardware enforcement
  – System-call interface inflexible
Type-safe programming language
  (e.g., Java)
  + Allows fine-grained data sharing
  – Not applicable to C/C++

SFI in outline
“Software-based Fault Isolation”
Simulate hardware-style protection with binary-level rewriting
Insert checks to confine jumps and memory writes to sandbox regions
Outline

SFI as a security technique
Classic (RISC) SFI
A CISC-compatible approach
PittSFIeld implementation
Machine-checked proof
Conclusion

Key problem: circumventing checks

f00: check %rs
f04: unsafe op %rs
f80: jmp f04
fbc: check-bounds %rt
fc0: jmp %rt

Do checks always precede unsafe ops?

Solution: dedicated registers

Indirect write only through %rs
Maintain invariant: at jump, %rs contains a legal data address
Safe to jump into middle of checks
f40: mov %rt -> %rs
f44: check %rs
f48: store %x, (%rs)
Requires several registers

Bitwise memory isolation

Distinct code and data areas to prevent self-modifying code
Areas have power-of-two size and alignment
Enforce by bitwise AND and OR on addresses

Ensure, don’t check

Ideal: if the original program would have violated the security policy, the transformed program will halt with an error message right before the violation.

Ensure, don’t check

Relaxed: if the original program would have violated the security policy, the transformed program will do something allowed by the security policy.
More optimizations

- Trusted register: check after modification, not before use
  - Invariant: frame pointer always safe for data region
- Guard pages: put unmapped pages at edges of data area
  - E.g., push needs no checks

Outline

- SFI as a security technique
- Classic (RISC) SFI
- A CISC-compatible approach
- PittSFIeld implementation
- Machine-checked proof
- Conclusion

Key problem: overlapping instructions

Key problem: overlapping instructions

Solution: enforce instruction alignment

- No instruction crosses a 16-byte boundary
- Jump targets have low 4 bits zero
- Call instructions end on 16-byte boundaries
- Only need one spare register

Optimization: AND-only sandboxing

Optimization: AND-only sandboxing

Security model

- Compiler and rewriter are untrusted
- Check rewriting on load; only this checker needs to be trusted
- Disallow unknown instructions
- Safety does not depend on compiler sanity
Outline

SFI as a security technique
Classic (RISC) SFI
A CISC-compatible approach
PittSField implementation
Machine-checked proof
Conclusion

Assembly-language rewriting

- Rewriter is a Perl program that operates on GAS assembly code
- Alignment using .align directives and conservative length estimation
- Important to rewrite before symbolic references resolved (done by code producer)

One-pass, local verification

- Single in-order pass over instruction sequence
- State machine keeps track of static invariant validity
  - Conservative assumptions at potential jump targets
  - Must clean up before jumping elsewhere

SPEC benchmarks (gcc = 1.0)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>time</th>
<th>size</th>
<th>compr. size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geom. Mean</td>
<td>1.21</td>
<td>1.75</td>
<td>1.07</td>
</tr>
<tr>
<td>164.gzip</td>
<td>1.16</td>
<td>1.65</td>
<td>1.10</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1.07</td>
<td>1.67</td>
<td>1.07</td>
</tr>
<tr>
<td>176.gcc</td>
<td>1.95</td>
<td>1.84</td>
<td>1.05</td>
</tr>
<tr>
<td>181.mlcf</td>
<td>1.01</td>
<td>1.74</td>
<td>1.13</td>
</tr>
<tr>
<td>186.mcfy</td>
<td>1.29</td>
<td>1.62</td>
<td>1.06</td>
</tr>
<tr>
<td>197.parser</td>
<td>1.14</td>
<td>1.92</td>
<td>1.06</td>
</tr>
<tr>
<td>252.eon</td>
<td>1.25</td>
<td>1.72</td>
<td>1.05</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1.36</td>
<td>1.96</td>
<td>1.07</td>
</tr>
<tr>
<td>254.gsp</td>
<td>1.24</td>
<td>1.84</td>
<td>1.05</td>
</tr>
<tr>
<td>255.vortex</td>
<td>1.23</td>
<td>1.63</td>
<td>0.98</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>1.16</td>
<td>1.63</td>
<td>1.09</td>
</tr>
<tr>
<td>300.twolf</td>
<td>1.08</td>
<td>1.80</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Sources of time overhead
Outline

SFI as a security technique
Classic (RISC) SFI
A CISC-compatible approach
PittSField implementation
Machine-checked proof
Conclusion

One good basket

- For security, key is verifier
- Want to know that if verifier says OK, code is really safe
- Prove it!
- Machine-checked proof for increased assurance

ACL2

- ACL2 is a proof-assistant environment from J Moore et al. (UT Austin)
- Model a problem in restricted subset of Common Lisp
  - (no mutation, higher-order functions)
- Refine goal into small sub-lemmas, each proved automatically (perhaps with ‘hints’)

Statement to prove

- Verifier implements a predicate on the code image
- Model the processor as an interpreter
- Unsafe operations cause it to halt, no exit
- \( \forall \text{ code}: (\text{code passes verifier}) \Rightarrow (\text{code runs forever}) \)

Proof status

- Verified for a small but representative instruction subset:
  
  \[
  \begin{align*}
  \text{nop} & \quad \text{mov} \quad \text{addr}, \%eax \quad \text{xchg} \quad \%eax, \%ebx \\
  \text{inc} \quad \%eax & \quad \text{mov} \quad \%eax, \quad \text{addr} \quad \text{xchg} \quad \%eax, \quad \%ebp \\
  \text{jmp} \quad \text{addr} & \quad \text{and} \quad \%immed, \quad \%ebx \quad \text{mov} \quad \%eax, \quad (\%ebx) \\
  \text{jmp} \quad \%ebx & \quad \text{and} \quad \%immed, \quad \%ebp \quad \text{mov} \quad \%eax, \quad (\%ebp) \\
  \end{align*}
  \]
- Realistic padding and encoding

Outline

SFI as a security technique
Classic (RISC) SFI
A CISC-compatible approach
PittSField implementation
Machine-checked proof
Conclusion
Conclusion

- It is possible to do SFI efficiently on a CISC architecture
- It is possible to apply SFI to full-scale applications
- It is possible to trust an SFI implementation