

Evaluating SFI for a CISC architecture

McCamant and Morrisett
presented by Cole Trapnell

SFI

A way of isolating (untrusted) modules by
instruction rewriting

SFI

- OS processes are a form of isolation
 - expensive
- Type-safe languages can also isolate untrusted code
 - requires using that language all the time
 - arguments for safety are less convincing than a machine-instruction-level technique

SFI

- “Sandboxes” code by rewriting, aligning the instruction stream
 - Idea due to Wahbe, Lucco, Anderson, and Graham
 - Key is prevent unsafe instructions (writes, jumps) from executing with bad args

Isolating pointers

1. Restrict pointers to regions with size = a power of 2
2. Align start of regions to that same power:
 - e.g. `0xda000000` – `0xdaffffff`
3. Now we can check operands with one or two instructions:
 - AND with `0xff000000` == `0xda000000`

Making checks unavoidable

- Idea: require sequential execution!
 - not practical, programs need jumps and branches...
- Direct branches and jumps are no problem, we can check those before execution
- Indirect jumps with a register operand are the real problem
 - returns, switches, function pointers, etc

Making checks unavoidable

- Want to limit jump target operands to safe instruction addresses, exclude unsafe instructions
 - Wahbe et al. direct unsafe operations through a dedicated register (`%rs`), effectively making all jumps safe
 - Writes to `%rs` are forced to have safe values

Contributions

New SFI technique, **PittSFIeld**, for CISC

1. Describes the technique and some new optimizations
2. Benchmarks it
3. Provides a use case study
4. Argues soundness via a machine-checked proof

CISC

- Really only talking about IA-32 (x86)
 - Variable length instructions
 - Instruction can start at any byte (no word alignment)
 - Few registers

PittSFieId instruction stream



Original instructions padded with no-ops to obey 16 byte chunk alignment

`call` instructions placed at the end of chunks

`jmp` instructions have low 4 bits zero

PittSFieId instruction stream

- It is impossible to execute the second instruction in a chunk without executing the first
- By making sure an unsafe instruction and its check can't be separated, **they never span chunk boundaries**

Optimizations

- Three previously described by Wahbe et al:
 1. Check `%ebp` only after modify instead of at each use as a data pointer
 2. Guard regions to avoid checks on load indirects using `%ebp`, `%esp`
 3. Ensure, don't check

Optimizations

- Two new ones:
 1. One-instruction address operations, since code and data region tags have only a single bit set
 - and `$0x20fffffff, %ebx`
 2. Efficient returns

Efficient returns

- Modern processors cache return addresses in a shadow stack.

Instead of this:

```
popl %ebx
and $0x10ffffff0, %ebx
jmp *%ebx
```

Use this:

```
and $0x10ffffff0, (%esp)
```

This optimization doesn't work if multiple threads may write to this address space

Verifier

- Rewritten code is checked just before execution
- Security policy is simple enough that verifier needs no help from the rewriter, and no access to symbol tables, etc.
- Checks the security property that a program never jumps outside code region or writes outside its data region

Rewriter implementation

- Register `%ebx` reserved and used to hold sandboxed addresses
- `%ebx` value checked before each write or indirect jump
- Rewrites input for the GNU assembler, not on off-the-shelf binaries

Rewriter implementation

- Alignment achieved via the `.p2align` directive of the GNU assembler
- Saves and restores status flags, but disables instruction scheduling.
 - Keeps checks of status flags close to corresponding branches

Rewriter implementation

```
push    %ebp
mov     %esp, %ebp
mov     8(%ebp), %edx
mov     48(%edx), %edx
lea    1(%eax), %ecx
```

```
mov     %ecx, (%ebx)
pop     %ebp
```

```
ret
```

```
push    %ebp
mov     %esp, %ebp
mov     8(%ebp), %edx
mov     48(%edx), %edx
lea    1(%eax), %ecx
```

```
lea    0(%esi), %esi
```

```
lea    48(%edx), %ebx
```

```
lea    0(%esi), %esi
```

```
lea    0(%edi), %edi
```

```
and    $0x20fffffff, %ebx
```

```
mov     %ecx, (%ebx)
```

```
pop     %ebp
```

```
lea    0(%esi), %esi
```

```
and    $0x20fffffff, %ebp
```

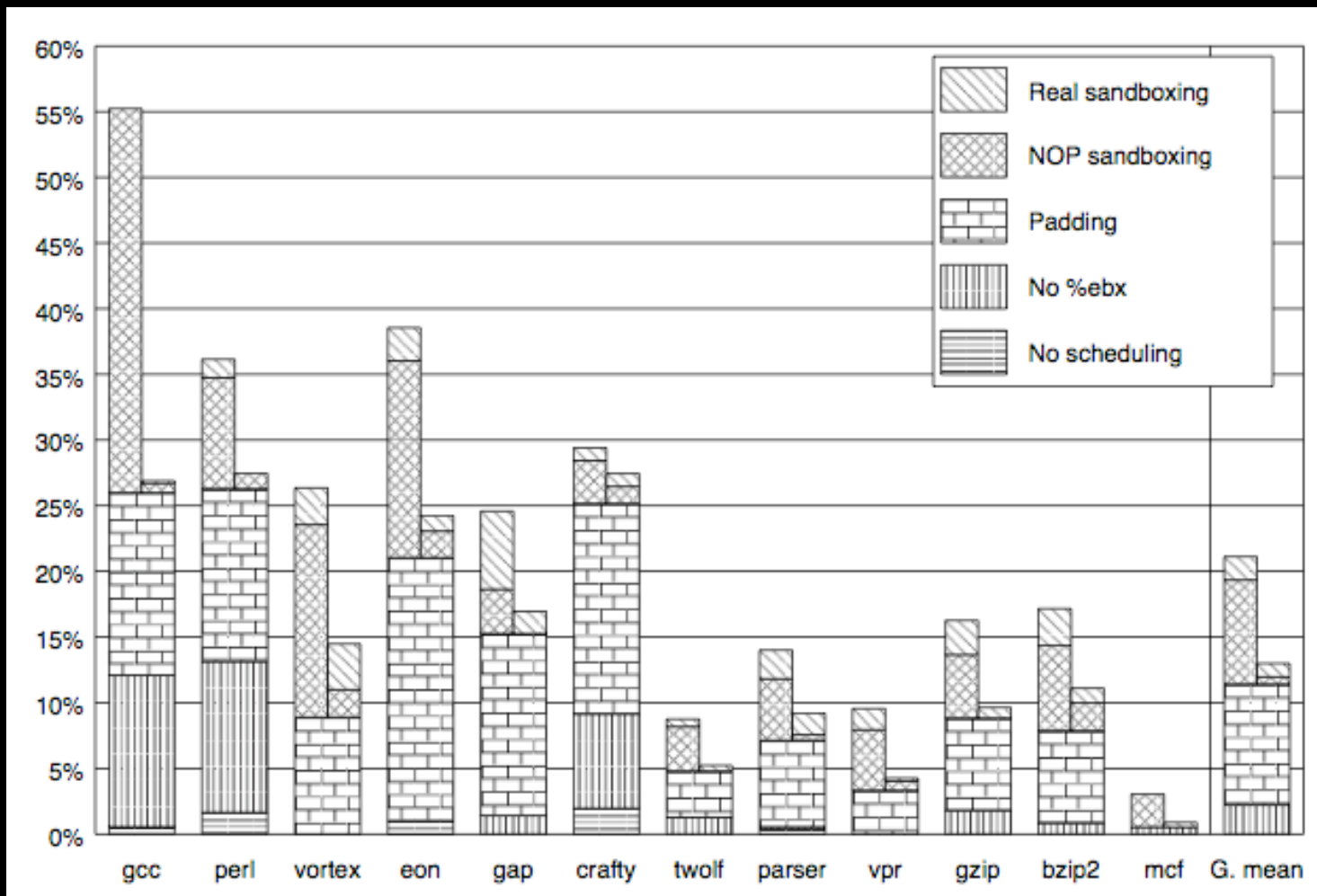
```
andl   $0x10fffff0, (%ebp)
```

```
ret
```

Verifier implementation

- In a single disassembly pass:
 - Checks the alignment requirement
 - Tracks deviations from the security invariant on a per instruction basis
 - Sandboxing ‘strengthens’ the invariant for one instruction or until a chunk is reached, whichever comes first
 - Ops that ‘weaken’ the invariant persist until corrected

Benchmarking performance



Overhead

- Rewritten SPECint2000 binaries are between 1.05- and 1.96-fold larger than the originals
- Compressed, rewritten binaries 0.98- and 1.10-fold larger than the compressed originals

Case study: VXA

- VXA is an archiving system in which 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

Formal analysis

- Proof of verifier's soundness is machine-checked with ACL2
 - ACL2 is a restricted subset of Common Lisp
 - Proof is a simplified model of the verifier, along with an x86 simulator
 - Proves the claim that if the verifier approves the rewritten code, it will only execute safe instructions, no matter the input state of memory/registers

Formal analysis

- No analysis of the rewriter, so no claim that rewritten code actually executes the behavior of the original program