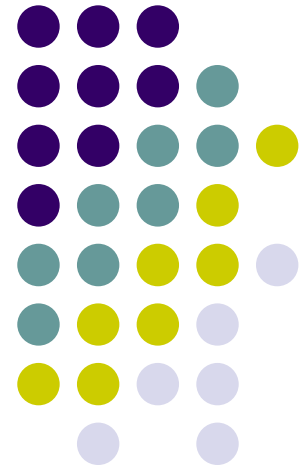


# Evaluating SFI for CISC Architecture

Richard Wartell

CS 6V81

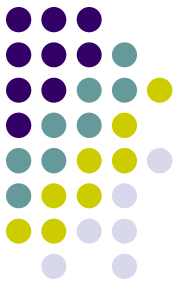
Language-Based Security





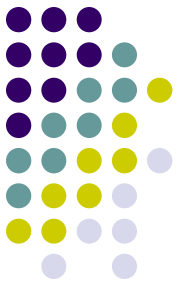
# What We Want to Achieve

- Code may come from an untrusted source, have been written for a malicious purpose, or have bugs that allow a malicious attacker to do undesired activities
- Trusted and untrusted code to be able to communicate, but not corrupt each other



# Other Methods

- Hardware Memory Protection
  - Code is restricted to accessing memory only in its address space.
  - Expensive due to interaction across a process boundary (system calls)



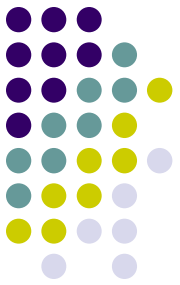
# Other Methods

- Type-safe Languages
  - Typing rules restrict code memory usage and control flow to safe patterns.
  - Solves the problem of interaction between separate code.
  - Usually for single language, unlike an OS
  - Difficult to make unsafe languages like C and C++ type-safe at all

# Sandboxing



- Rewrite machine code to enforce memory and control flow policies.
- Many systems can benefit from this.
- Very low runtime overheads.

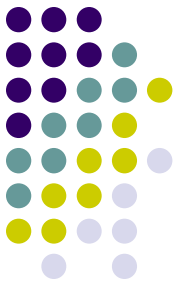


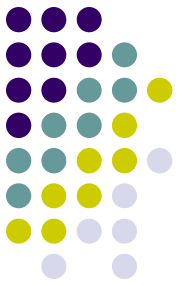
# The Paper

- Their approach to SFI
- Verification technique
- Optimizations
- Benchmarks
- How embedded decompression can be accomplished
- Proof of soundness

# Classic SFI

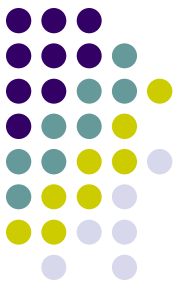
- Discussed in the last paper





# SFI for CISC

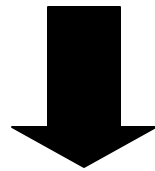
- Classic SFI does not immediately apply to CISC architectures, such as x86, due to variable-length instructions.
- Restricting jumps to a single stream of instructions won't work.
- X86 instructions may start at any byte



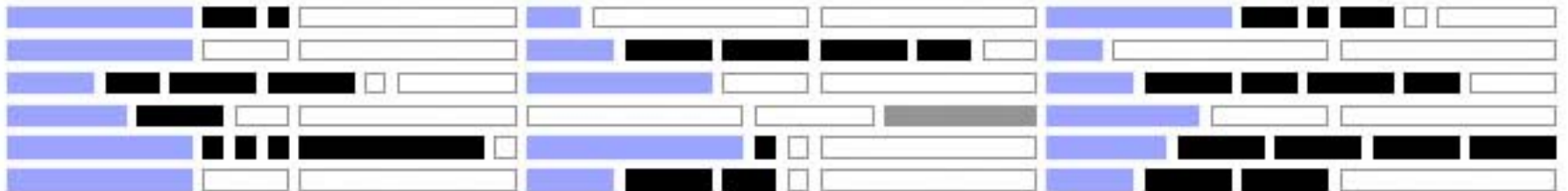
# SFI for CISC

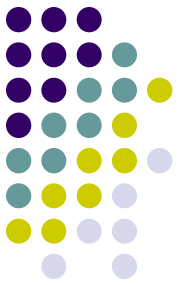
- The Solution
  - Divide memory into “chunks” (16 bytes)
  - Insert no-ops so that no instruction crosses a chunk boundary.
  - Targets are put at the beginning of chunks.
  - Calls go at the end of chunks.
  - Jumps have their low 4 bits zero.
  - Unsafe operations and the check of its operand cannot be separated or cross chunk boundaries

# Example



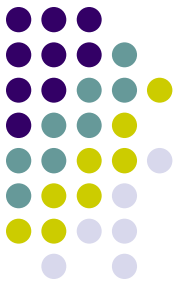
PittSFeld  
Rewriting





# Verification

- Tool is created in two separate pieces
  - Program Rewriter
    - Rewrites the code to conform to memory safety and control flow safety
  - Verifier
    - Enforces safety policies
    - Checks the rewritten code prior to execution



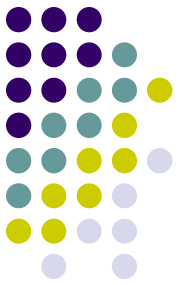
# Verification

- Conservatively checks properties
  - If true, the code will not violate the system's security policies
- At each position in the instruction stream
  - The content of the processor's registers for any execution
  - If property holds for all positions, then the code is verified

# PittSField

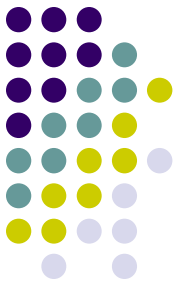


- On the Web
- PittSField
  - Includes major optimizations
  - Separated code rewriting and verification
  - All of the fundamentals of the idea included
  - Not fully optimized
  - Gives a good upper bound



# PittSField

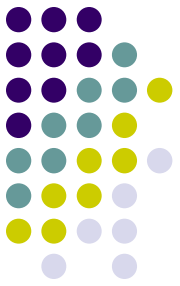
- Intended to be used by code producer
- Top-to-bottom executions
- Written in PERL
- A second verifier was written using a pre-existing disassembly library
  - Executes at runtime prior to code execution



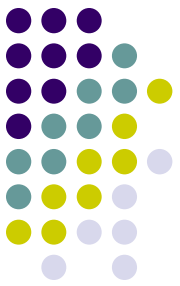
# Optimizations

- Large performance penalty introduced by this system
- Many optimizations exist
  - Reduces the overhead of rewriting
  - Makes it more complex

# Special Registers

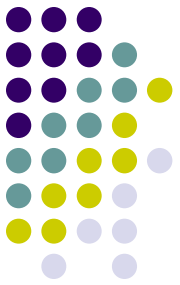


- Frame pointer
  - Usually set at the start of a function
  - Used after that
  - Check only after modification, rather than at each use



# Guard Regions

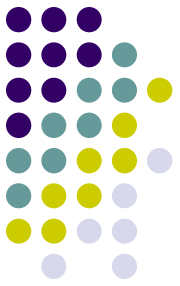
- Areas before and after the data region that the sandboxed code can write to safely
  - Accesses to these regions are efficiently trapped
- Same as previous paper
- Small modifications of the stack pointer that are not used to read or write can be a problem
  - Check a modified stack pointer prior to a jump



# Ensure, Don't Check

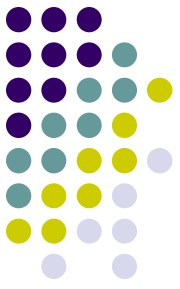
- What happens when untrusted code attempts an unsafe operation?
  - Terminate the code
  - Or
  - Perform a safe action instead
    - Set bits to an arbitrary other location in the code region
    - More efficient than a system call
    - This could be unhelpful for some programs (i.e. debugging)

# One-instruction address operations

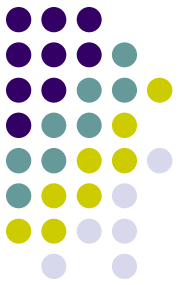


- Usually for a data region, bits must be cleared and set
  - This requires an AND and an OR operation
- Restricting regions can allow you to only need to clear bits via an AND operation

# One-instruction address operations

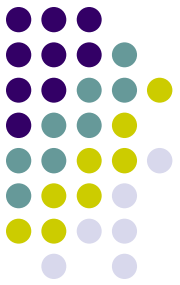


- Eg.
  - 16mb code and data regions, starting at 0x01000000 and 0x02000000 respectively
  - To ensure that an address is legal
    - and \$0x20FFFFFF, %ebx
    - The address is then either in the data region, or the zero tag region, 0x00
  - Cuts overhead by 10%



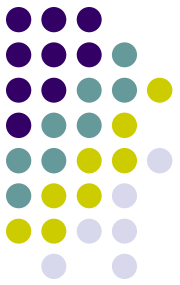
# Efficient Returns

- A ‘branch target buffer’ is kept of the most recent target for a jump instruction
  - A naïve implementation would use this to handle procedure returns
    - `popl %ebx`  
and `$0x10FFFFFF0, %ebx`  
`jmp *%ebx`



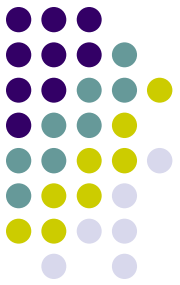
# Efficient Returns

- Doesn't work if procedure is called from multiple locations
- Instead use a real ret instruction and the shadow stack of return addresses
  - Doesn't need a scratch buffer
    - and \$0x10ffff0, (%esp)  
ret
    - Worst case Fibonacci, overhead reduced from 95% to 40%.
    - An average increase in overhead of 6% if not used



# Formal Analysis

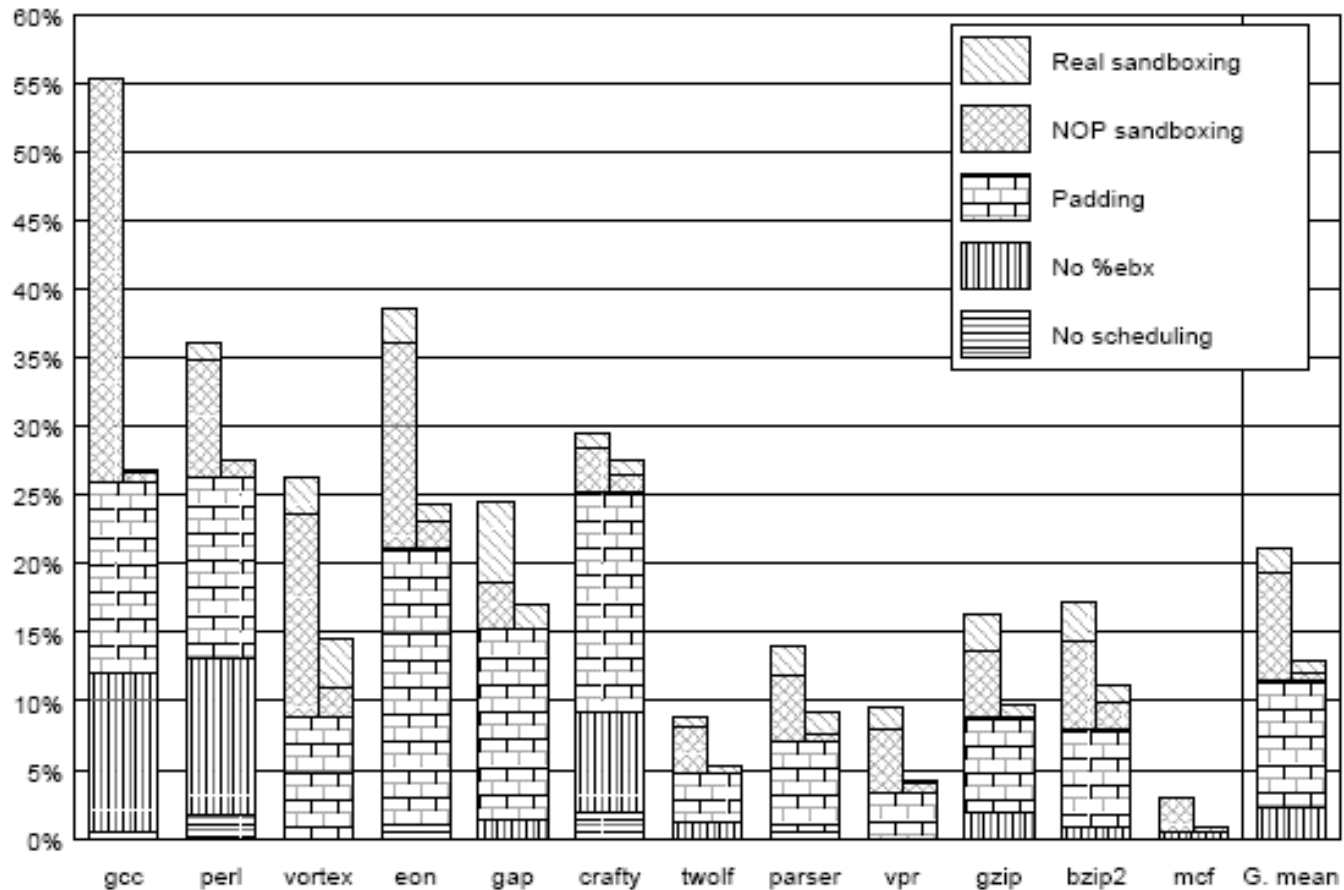
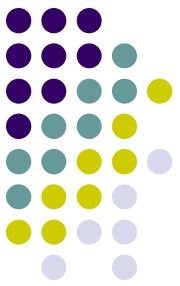
- Proof given in ACL2
- Was compiled by an user in less than 2 months with no previous ACL2 experience
- The subset of codes they used:
  - `nop`            `mov addr, %eax`            `xchg %eax, %ebx`
  - `inc %eax`        `mov %eax, addr`            `xchg %eax, %ebp`
  - `jmp addr`        `and $immed, %ebx`        `mov %eax, (%ebx)`
  - `jmp *%ebx`        `and $immed, %ebp`        `mov %eax, (%ebp)`
  - All other instructions cause an immediate failure

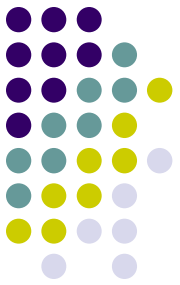


# Performance

- Ran the software on a series of 12 programs
- 1 written in C++, the rest in C
- View as an upper bound

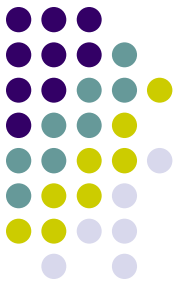
# Performance





# Related Work

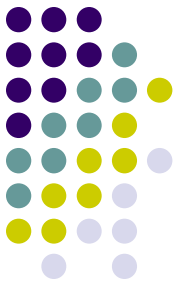
- Decompression Model
- Other SFI implementations
- Isolation and preventing subversion
- CFI
- Static C Safety Mechanisms
- Dynamic Translation Mechanisms
- Low-level Type Safety



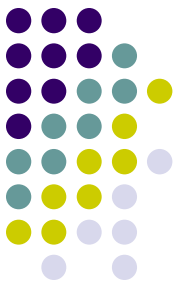
# Future Problems

- Further Optimization
- Complete Proof of All x86 Instructions
- Integration with Proof-Carrying Code
- Code User Implementation
- Others?

# References



- Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, August 2006.
- Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203-216, December 1993.



# Questions

- Does the PittSField implementation cause an exponential growth in overhead?
- Is it possible to modify this implementation so that it would be feasible for a consumer to use it to enforce their own security policies?
- Questions or Comments?