Efficient Software-Based Fault Isolation

Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham

Problem and Motivation

- Application developers frequently incorporate other code bases into their own applications
  - E.g. it’s easier (and usually safer) to someone else’s crypto library than to roll your own
- How can I (an application developer) mitigate negative side effects of third-party code used by my application?

Fault Isolation (FI)

- Isolate components of a system, so that if a fault occurs in one of the isolated components:
  - The side effects of that fault are constrained to the isolated environment
  - It is easy to track down the faulting component and pinpoint the exact cause
- Existing solutions place components in their own address space
  - Depends on hardware to enforce isolation
- Address-space based isolation incurs a significant performance penalty
  - Crossing isolation environments triggers a context switch
  - Context switches are expensive
  - Lots of context switches = lots of overhead

Software Fault Isolation (SFI)

- Load untrusted component into its own memory region
- Ensure that code executed in such a region cannot write or transfer control to memory addresses outside of said memory region
- Advantages
  - Isolation of untrusted components
  - No context switch incurred to communicate between safe-unsafe and unsafe-unsafe, since all sandboxes exist in the same address space

Static or Dynamic Instrumentation

- SFI can be performed both statically and dynamically
- Dynamic
  - Code has already been compiled
  - SFI instrumentation is added via dynamic binary rewriting (“binary patching”)
- Static
  - Compiler-based instrumentation
  - Binaries are generated with SFI enforcement
- Paper chose the static route

Methodology

- Divide an application’s virtual address space in segments
- Segments are aligned in such a way that the high bits of an address in that segment are unique to that segment (forms a segment ID)
- Each segment is further split into two regions:
  - Code
  - Data
    - Stack, heap, static data, etc.
- Code is added that checks the target address of writes and control transfers (Segment Matching)?
  - Does the segment ID of the target address match the segment ID of the current segment?

Implementation (Generate Unsafe Instruction Set)

- Use static analysis to reduce the set of instructions that need to be instrumented
  - Possible to statically verify most uses of memory address in write and control transfers
    - Uses of memory derived from offsets relative to the program counter can be checked to be in bounds at compile time
    - Uses of absolute memory addresses
      - E.g. direct calls and jumps
- Indirect writes and control transfers make up the majority of the unsafe instruction set
  - E.g. uses of addresses stored in registers

Implementation (Instrumentation)

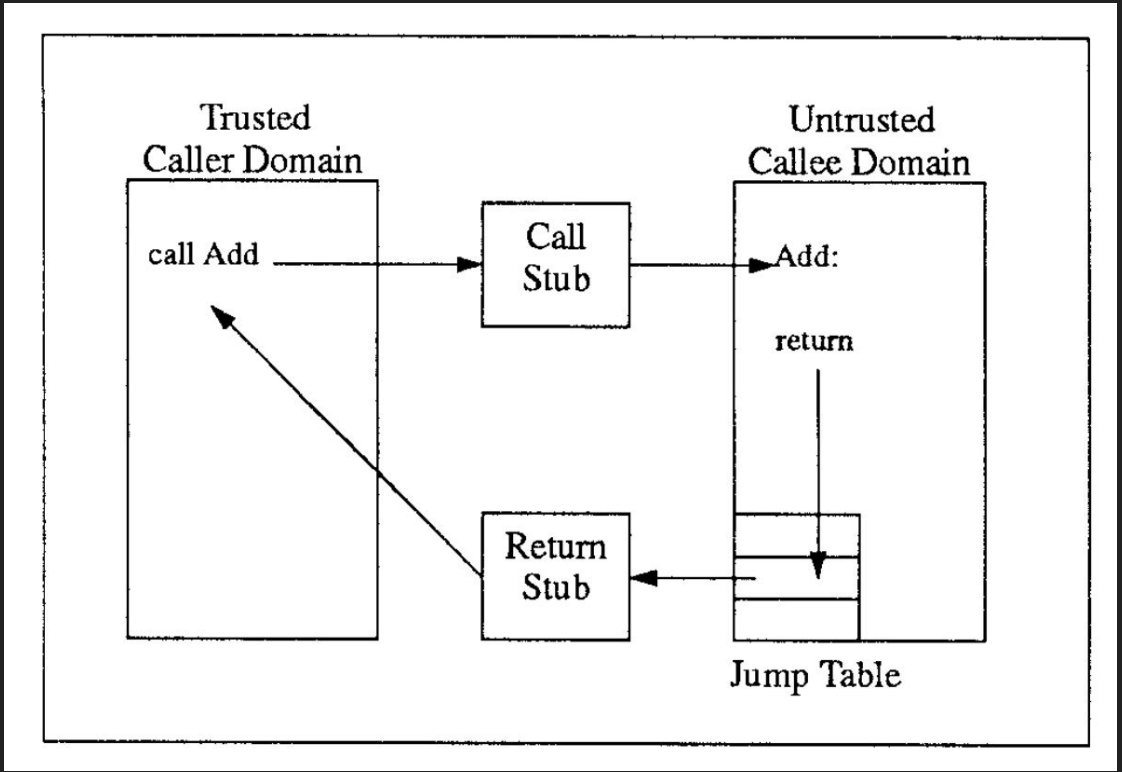
```
dedicated-reg ⇐ target address
    Move target address into dedicated register.
scratch-reg ⇐ (dedicated-reg>>shift-reg)
    Right-shift address to get segment identifier.
    scratch-reg is not a dedicated register.
    shift-reg is a dedicated register.
compare scratch-reg and segment-reg
    segment-reg is a dedicated register.
trap if not equal
    Trap if store address is outside of segment.
store instruction uses dedicated-reg
```

Segment Matching

```
dedicated-reg ⇐ target-reg&and-mask-reg
    Use dedicated register and-mask-reg
    to clear segment identifier bits.
dedicated-reg ⇐ dedicated-reg|segment-reg
    Use dedicated register segment-reg
    to set segment identifier bits.
store instruction uses dedicated-reg
```

Sandboxing

Implementation (Cross-Segment RPC)



- Custom call and return stubs are created for each exported function in the untrusted callee domain
- Destinations in jump table entries are encoded in the instruction
- Jump table is stored in read-only memory
- Call stub copies arguments directly in untrusted callee domain

Implementation (System Resource Usage)

- What if code in an unsafe region needs to access a system resource?
  - E.g. the unsafe code needs to make a syscall
- All accesses to global resources are routed through an “arbitration code” that exists in its own segment
  - Requests for access are passed through cross-domain RPC calls
- If allowed, arbitration code performs the requested action on behalf of the unsafe code, and returns the result (if any)

Evaluation

- Software Encapsulation Overhead
  - Used the sandboxing primitive
  - Benchmarking code was added to an existing code base, but placed inside of an unsafe region
  - ~4.3% on average across all benchmarks
- RPC Overhead
  - Tested with an instrumented version of POSTGRES
  - Overhead measure between ~1.7 and 5.7% across all benchmarks
  - RPC overhead (context switching) estimated to be between ~18.6 and 38.6%

Key Takeaways

- SFI can be used to create isolated regions within the same virtual address space
- SFI is, on average, more performant than FI with hardware-based RPC
    - Author admits that kernel optimizations can be made to reduce the cost of hardware-based RPC, but as long as overhead is above 5%, SFI should perform better

Questions

- There are two SFI implementations described in this paper: binary patching (dynamic) and compiler-enforced (static). Is the dynamic approach possible for x86/64?
  - Disassembling x86/64 is imprecise
  - What about misaligned instructions?

Questions

- If we have to use a compiler-based implementation, is the efficacy of SFI dependant on its adoption rate by writers of third-party libraries?
  - Most developers don't compile their own versions of libraries and carry them around; they typically use shared ones
  - What if a shared library wasn't compiled with SFI enabled?
  - ASLR 2.0?

Questions

- Does the SFI model work if it is dependent on developer adoption?
  - The idea behind SFI is that you, the owner of your code, decide what modules are "unsafe"
    - If library developer adoption is required, then that decision is taken out of the hands of consumers
  - As a library developer, do I have to assume that my code is “unsafe” and incur the performance penalty?